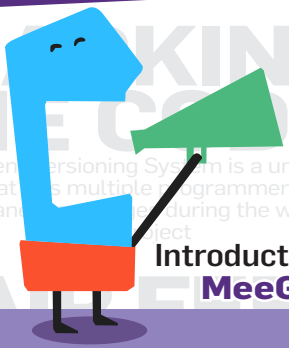
VOLUME 06 | ISSUE 06B

digit FastTrack

YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

To MeeGo



Introduction to MeeGo

MeeGo development environment

Introduction to Qt

Build MeeGo apps for AppUp

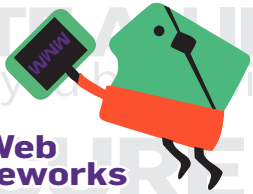
Working with Qt creator

How to create UI layouts

Developing MeeGo apps with Python

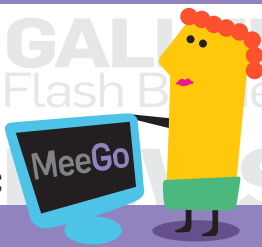
Introduction to QML

Tips



Web frameworks

Install MeeGo on your PC



A 9.9 Media Publication

MEEGO

powered by

CHAPTERS

MEEGO | JUNE 2011

CREDITS

The People Behind This Book

EDITORIAL

Editor
Robert Sovereign-Smith

Head-Copy Desk
Nash David

Writers
Ankur Mour, Meghnil Pagrut

DESIGN

Sr. Creative Director
Jayan K Narayanan

Art Director
Binesh Sreedharan

Associate Art Director
Anil VK

Sr. Visualisers
PC Anoop

Senior Designers
Baiju N V, Chander Dange, Vinod Shinde

COVER DESIGN BINESH SREEDHARAN

June 2011
Free with Digit. Not to be sold separately. If you have paid separately for this book, please email the editor at editor@thinkdigit.com along with details of location of purchase, for appropriate action.

INTRODUCTION

MeeGo traces its genesis to the February 15, 2010 announcement, where Intel, the world's largest chip manufacturer, announced a collaboration over their existing open source projects (Moblin and Maemo respectively) to form a new project called MeeGo, hosted at the Linux Foundation. According to Intel, MeeGo was developed because Microsoft did not offer comprehensive Windows 7 support for the Atom processor.

We've tried to present you with a comprehensive guide to developing applications in MeeGo, an operating system by Intel designed with the Atom processor in mind.

In chapter 1, we take a closer look at the OS, its structure and some finer detail that would help you gain a better understanding of how it works.

In chapter 2, we have covered the prerequisites you would have to consider before you take a stab at developing an app in MeeGo.

We then talk about the individual tools needed to build apps. Chapter 3 talks about Qt (pronounced cute) and not cue-tee. Qt is the framework used to develop applications in MeeGo.

Chapter 4 talks about Qt Creator which is the IDE (integrated development environment) similar to Eclipse that is otherwise popularly used to develop applications.

Chapter 5 looks at different user interfaces and guides you on how to create them.

Chapter 6 is an introduction to QML, which decides the look and feel of the user interface.

In chapter 7 we look at developing MeeGo apps using Python. Python is a powerful language used by Google for some of its key applications.

Chapter 8 looks at the Intel AppUp℠ centre, which is Intel's version of an app store, a concept popularised by Apple.

In addition to the Fast Track, we have two bonus chapter on web frameworks and another that talks about installing MeeGo on your desktop PC. These bonus chapters are included on this month's DVD.

We wish you all the best in your attempt at building applications for the MeeGo platform! d

SCAN THE **QR CODE** AND GET ALL THE URLs FROM THIS FAST TRACK ON YOUR SMARTPHONE

MEEGO

A brief overview of the MeeGo architecture, its benefits and the derivatives that came out of Intel's operating system for portable systems.

MeeGo is an open source operating system for the next generation of computing devices. It's primarily targeted at mobile devices and appliances in the consumer electronics market and is designed to act as an operating system for a wide range of hardware platforms such as netbooks, entry-level desktops, tablet computers, in-vehicle infotainment devices and other embedded systems.

Nokia Maemo

Nokia's Maemo project provided a Linux-based software stack that runs on mobile devices. The Maemo platform is built using open source components and its SDK provides an open development environment for applications.

A series of Nokia Internet Tablets with touch screen have been built with the Maemo platform.

Intel Moblin

The Moblin (Mobile Linux) project is Intel's open source initiative created to develop software for smartphones, netbooks, and other mobile devices. It is optimised for small computing devices and runs on the Intel Atom processor.

User interfaces

Within the MeeGo project there are several graphical user interfaces as follows:

Netbook UX

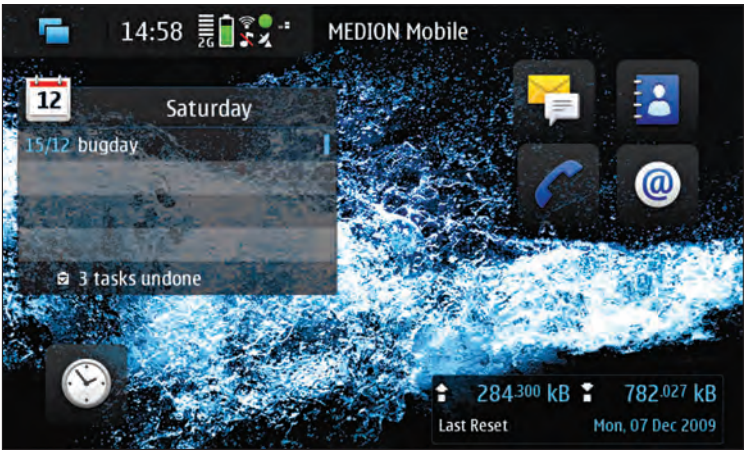
The netbook is the first user experience to become available for MeeGo. After being released on May 25, 2010, it delivered a wealth of computing and communication experiences with rich graphics, multi-tasking and multimedia capabilities, and is highly optimised for power and performance. The netbook UX is a continuation of the Moblin interface. It is written using the clutter-based Mx toolkit, and uses the Mutter window manager.

MeeGo's netbook version integrated several Linux applications in the background, such as Evolution (Email, calendar), Empathy (instant messaging), Gwibber (microblogging), Chromium (web browser), and Banshee (multimedia player).

Handset UX

The Handset UX is based on Qt, but GTK+ and Clutter have been included to provide compatibility for Moblin applications. Applications will be provided from either the Intel AppUp or the Nokia Ovi digital software distribution systems depending on which device you use.

Version 1.1 was released in October 2010 and included key handset technologies such as cellular, connectivity, sensors, and mobile browsing, as well as



The Netbook UX for Moblin is optimised for portable computers

a basic development UX for voice calling, SMS messaging, web browsing, music and video playback, photo viewing, and connection management.

Tablet UX

In February 2011, the chipmaker released, what it called, a "developer preview" of the MeeGo user experience (or "UX") for tablet computers. In April, it officially made the source code available too, making it one of the first publicly-visible

Handheld OS?: MeeGo even has a handheld UX

glimpses at MeeGo 1.2.

The Tablet UX borrows a lot from the Handset UX in terms of layout, starting with the white-icon-on-black taskbar that is now common on every mobile platform and is making its way onto Linux desktop environments as well. The Tablet UX shows a small, but highly functional suite of apps, and it wraps it up in a slick and user-friendly package.

In-Vehicle Infotainment

In-Vehicle Infotainment (IVI) are navigation, entertainment and networking systems providing computing services in vehicles, such as cars, trucks, planes, and buses. The MeeGo IVI software platform is designed to enable rich internet and multimedia consumer experiences for vehicles. MeeGo IVI builds on the foundation laid by Moblin IVI for rich multimedia, CE device management, internet, and automotive connectivity.

Technical Stuff

Core OS

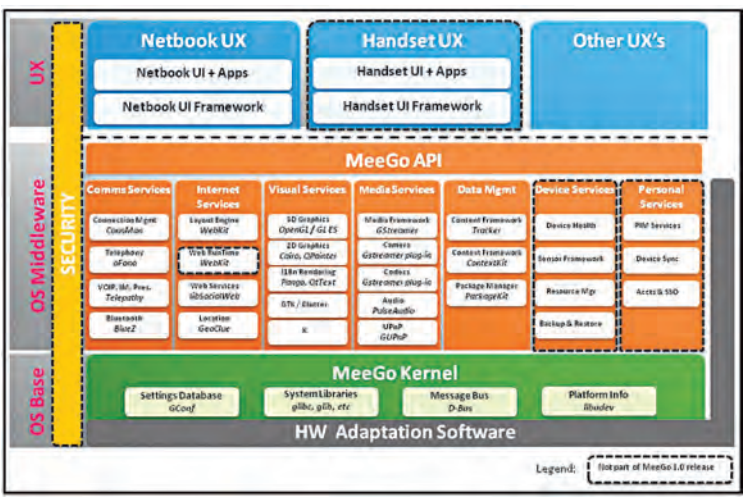
The MeeGo Core operating system is a Linux distribution, drawing on Nokia's Debian-based Maemo and Intel's Fedora-based Moblin. MeeGo is one of the first Linux distributions to use the Btrfs file system as default, and uses RPM repositories.

MeeGo's prepared to take on tablets too

Architecture

MeeGo has been designed to offer the best device user experience. The MeeGo platform architecture can be viewed in three different ways:

- **Layer view** shows the separation of different layers and user experience (UX) verticals
- **Domain view** shows the grouping of subsystems into architecture domains, based on similarities in technology and functionality
- **API view** shows the grouping of MeeGo API into functional area



MeeGo has distinct UXs for individual product categories, and a supporting API and kernel

Layer View

The MeeGo architecture as divided into three layers:

- The MeeGo OS Base layer consists of the Hardware Adaptation Software required to adapt MeeGo to support various hardware architectures and the Linux kernel and core services
- The MeeGo OS Middleware layer provides a hardware and usage model independent API for building both native applications and web run time applications.
- The MeeGo User Experience (UX) layer provides reference user experiences for multiple platform segments. The first UX reference implementation was released on May 25, 2010 and it was for the netbook UX. Other UX reference implementation will follow for additional supported device types.

Domain View

The Domain view expands each domain and details the subsystems required to provide that functionality.

- **Security domain is responsible of security deployment across the system. It provides enablers for platform security and user identity.**
- **Data Management** domain provides services for extracting and managing file meta-data (for example to support extracting and searching metadata for media files).
- **Software Management** domain is responsible for package manager and its backend functionality.
- **System** domain is responsible for device state/mode handling, time management, policy control, startup services, and sensor abstraction.
- **Location** domain provides location services.
- **Kernel** domain contains Linux kernel and device drivers.
- **Personal Information Management** domain enables managing user data on the device, including managing calendar, contacts, tasks, and retrieving data about the device context (such as device position, cable status).
- **Multimedia** domain provides audio and video playback, streaming, and imaging functionality to the system. In general, the domain takes care of the actual audio and video data handling (retrieval, demuxing, decoding and encoding, seeking, etc.).
- **Essentials** domain provides all system essential packages.
- **Communications** domain provides Cellular and IP Telephony, Instant Messaging, Presence, Bluetooth, and Internet Connectivity services.
- **Qt** domain contains cross platform toolkits such as Qt, Qt Mobility, Qt WebKit, and Qt WebRuntime.

- **Graphics** domain enables the core 2D and 3D graphics capabilities for the platform, including support for rendering internationalized text and taking advantage of underlying hardware platform acceleration for graphics.
- **Change History** provides what has changed since the original creation of page on October 22, 2010.

API View

The API view illustrates the contents of MeeGo API.

MeeGo API is based on Qt and Qt Mobility.

Qt

Qt provides application developers with the functionality to build applications with state-of-the-art graphical user interfaces. Qt is fully object-oriented, easily extensible, and allows true component programming.

Qt mobility

Qt Mobility delivers a set of APIs to Qt, with features that are well known from the mobile device world. However, these APIs allow the developer to, with ease, use features from one framework and apply them to phones, netbooks, and non-mobile personal computers.

Derivatives

Along with Moblin before, MeeGo provides a huge technology pool so that software vendors can access and build their products from. So far only ports of the graphical user interfaces to other Linux distributions have been announced.

MeeGo/Harmattan

Although MeeGo was initiated as a collaboration between Intel and another handset manufacturer, the collaboration was formed when the handset manufacturer was already developing the next incarnation of its Maemo Linux distribution. As a result, the Maemo 6 base operating system will be kept intact while the Handset UX will be shared, with the name changed to "MeeGo/Harmattan".

SUSE and Smeegol Linux

On June 1, 2010 Novell announced that they would ship a SUSE Linux incarnation with MeeGo's Netbook UX (MeeGo User Experience) graphical user interface. A MeeGo-based Linux distribution with this user interface is already available from openSUSE's Goblin Team under the name Smeegol Linux, this

project combines MeeGo with openSUSE to get a new netbook-designed Linux distribution.

Fedora

Fedora 14 contains a selection of software from the MeeGo project.

Linpus

Linpus Technologies is working on bringing their services on top of MeeGo Netbook and MeeGo Tablet.

Splashtop

The latest version of the instant-on OS Splashtop-platform (by Splashtop Inc. which was previously named DeviceVM Inc.) is compliant with MeeGo, and future version of Splashtop will be based on MeeGo and will be available for commercial use in the first half of 2011.

Releases

After announcing at the Intel Developer Forum 2010 that MeeGo would follow a six month release schedule. Each release cycle will follow the same basic pattern of phases as described below. Each release will contain the MeeGo core and the existing set of MeeGo categories.

The MeeGo Project has had multiple releases and has progressed significantly since its announcement in February 2010. MeeGo has delivered the core software platform in addition to three user experience implementations (Netbook, handset and in-vehicle infotainment), with several updates in between.

Along with major releases, MeeGo offers updates that usually include general operating system fixes to enhance the stability, compatibility, security and visual quality of the devices running MeeGo. Between MeeGo 1.0 (05/2010) and MeeGo 1.1 (10/2010), the MeeGo Project provided three update releases that featured improvements to the MeeGo core stack and the Netbook release. The latest version released was MeeGo 1.1.3 in February 2011 which Fixed many important security issues, enabled all programs to access remote files over network and updated translation.

Benefits of the MeeGo Software Platform

Everyone in the ecosystem starting from the developer all the way up to the operator and the industry as a whole benefits form the MeeGo open source project. Meego allows participants to get involved and contribute to an industry-

wide evolution towards richer devices, to rapidly address opportunities and to focus on differentiation in their target markets.

Open Source Developers

The MeeGo project is a true open source project hosted by the Linux Foundation and governed by best practices of open source development. From meego.com, as an open source developer, you have access to tools, mailing lists, discussion forum, accessibility to technical meetings, and multiple options to make your voice heard over technical and non-technical MeeGo related topics. Furthermore, all source code contributions needed for MeeGo will be submitted to the upstream open source projects from which MeeGo will be built.

Application developers

MeeGo greatly expands the market opportunities for you being the only open source software platform that supports deployments across many computing device types. MeeGo offers Qt and Web runtime for application development, cross platform environments, so application developers can write their applications once and deploy easily on many types of MeeGo devices or even on other platforms supporting the same development environment.

In addition, MeeGo application developers will the opportunity to make their applications available from multiple application stores such as the Nokia's Ovi Store (*https://store.ovi.com*) and the Intel's AppUp Center (*http://www.intel.com/consumer/products/appup.htm*).

Device manufacturers

MeeGo helps accelerate time to market using an off-the-shelf, open source and optimised software stack targeted for the specific hardware architecture the device manufacturer is supporting. This helps the device manufacturer as MeeGo lowers complexities involved in targeting multiple device segments by allowing the use of the same software platform for different client devices. Also, as an open source project, MeeGo enables device manufacturers to participate in the evolution of the software platform and build their own assets for it through the open development model.

Operators

For operators, MeeGo enables differentiation through user interface customization. Although many devices can be running the same base software platform, they can all have different user experiences. Furthermore, it provides a single

platform for multitude of devices, minimizing the efforts needed by the operators in training their teams and allows their subscribers to be familiar with the experience common to many device types.

Linux platform

In addition, MeeGo is helpful for Linux as a platform as it combines mobile development resources that were recently split in the Maemo and Moblin projects into one well-supported, well-designed project that addresses cross-platform, cross-device and cross-architecture development. One major benefit from the MeeGo project is that all other Linux mobile and desktop efforts that use the components as MeeGo will benefit from the increased engineering efforts on those components.

10 facts you must know about MeeGo

1. Full open source project governed according to best practices of open source development: Open discussion forums, open mailing lists, open technical steering committee meetings, peer review, open bugzilla, etc.
2. MeeGo has no contributors' agreements to sign; instead it follows the same "signed-off-by" language and process as the Linux Kernel.
3. It enables all players of the industry to participate in the evolution of the software platform and to build their own assets on MeeGo
4. It offers a compliance program to ensure API and ABI compatibility and to certify software stacks and application portability.
5. MeeGo is aligned closely with upstream projects and it requires that submitted patches also be submitted to the appropriate upstream projects and be on a path for acceptance.
6. It supports multiple hardware architectures and multiple app stores.
7. Meego lowers complexity for targeting multiple device segments
8. Offers differentiation abilities through user experience customisation
9. Has over 1000 committed professional developers and hundreds of open source developers and very a vibrant community of users and developers (~ 8000 subscribed to meego.com)
10. MeeGo 1.0 Netbook release supports the following languages: Japanese, Korean, Chinese Simplified, Chinese Traditional, Swedish, Polish, Finnish, Italian, Brazilian Portuguese, French, German, Spanish, Russian, Dutch, English, and British English.

SETTING UP YOUR MEEGO DEVELOPMENT ENVIRONMENT

You've got an idea of what to expect from MeeGo. Now try your hand at developing some MeeGo apps. What would you need for that? Read on.

The MeeGo developer is based on the Qt SDK and API and it contains includes the most recent releases of Qt Quick and Qt Mobility. With this powerful platform and the extensive documentation available, you can create Qt applications that target the different MeeGo UXs. From the MeeGo SDK you will be able to control the full application development cycle.

MeeGo v1.2 SDK

MeeGo v1.2 SDK release provides a Qt Creator-based development environment for developing, debugging, and running mobile applications on

N900, Aava and Netbook devices. An emulator (QEMU) is also provided for developing without target hardware.

MeeGo SDK components:

- QtCreator 2.0.1
  - qt-tools 4.7.0
- Toolchains
  - arm toolchain (cs2009q1 based)
  - ia32 toolchain (meego based)
- MADDE 0.7.53
  - core sysroot (armv7l, ia32)
- Emulator
  - qemu-gl
  - QEMU images
    - core (armv7l, ia32)
    - handset (armv7l, ia32)
    - netbook (ia32)

Getting started with the MeeGo SDK for Windows

Introduction

Many of you experienced Windows developers are looking for ways to leverage their skills in new mobile markets. This MeeGo SDK for Windows enables developers to build applications for MeeGo, using the environment and tools with which they are most effective.

To avoid confusion, you should be aware that although Qt Creator is available on Windows from Nokia, only the version of Qt Creator that is included with the MeeGo SDK for Windows is set up to cross compile from Windows to MeeGo out of the box.

The SDK supports developing for the MeeGo tablet, netbook, and handset profiles, including emulators for each environment.

Prerequisites

- **Hardware:** Any reasonably modern IA hardware, such as 32-bit Intel Atom or Intel Core 2 CPU.
- **Known to work:** Intel(R) Core(TM) 2 Quad CPU, Q8200; Intel(R) i5 and i7
- **Known not to work:** Intel(R) Core(TM) 2 Duo P9400 in Thinkpad T400; Intel(R) Core(TM)2 Q9400 in HP7900 Ultra-slim
- **Software:** Only Windows XP 32-bit and Windows 7 32-bit are officially supported. Currently the QEMU emulator only works for IA targets on

32-bit Windows. If you are using 64-bit Windows, QEMU is not available.

- **Disk space:** At least 3 GB of free disk space is needed.
- **Filesystem:** MeeGo SDK should be installed on a hardlink-capable filesystem such as NTFS. FAT* filesystems are not hardlink-capable and not supported.

A Brief Walk-Through of the Steps

To help you understand what you need to do and where this section is going, here is a brief explanation of the steps needed to install the MeeGo SDK.

The first step is to download and launch the MeeGo SDK's install Wizard. This will allow you to select components that you want to install and will copy them to the correct directories on your system.

The installation process has been greatly simplified in the new Meego v1.2 SDK. This takes care of downloading files such as toolchains, build libraries, and virtual machine images. If there are errors on the network and the transfers fail, click on "Retry" when the installer tries to recover. This will most often result in successful continuation. However if this does not work, click "Ignore". This will skip the file and the SDK will be incomplete.

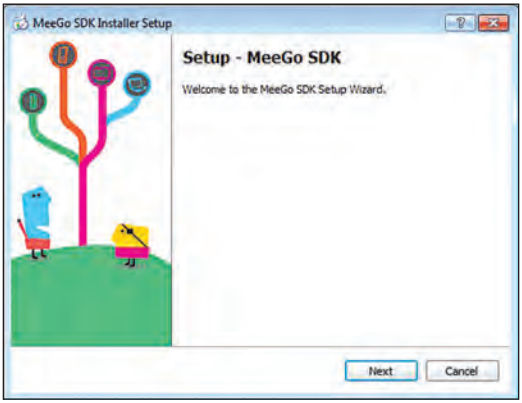
Now we configure Qt Creator, the MeeGo development IDE, to either run the QEMU emulator or set up a remote deployment device. The remote deployment device can be any MeeGo device such as a netbook or tablet. Deployments to the remote device are done over the network, and you will need to know the IP address of your remote device.

Hopefully that gives you a good overview of what we are trying to accomplish. Now on to work...

Download and install the MeeGo SDK Tools

- Download the MeeGo SDK Windows Installer (~54 Mb) from *http://developer.meego.com/meego-sdk.*
- Double-click on the downloaded .exe file and follow the prompts to install the SDK.

Note : If you are using Windows XP, please download a small file vcredist_x86.exe (Microsoft Visual C++ 2008 SP1 Redistributable Package (x86)) from *http://www.microsoft.com/downloads/en/details.aspx?FamilyID=a5c84275-3b97-4ab7-a40d-3802b2af5fc2* and install it to your system. This is not needed for Windows 7.



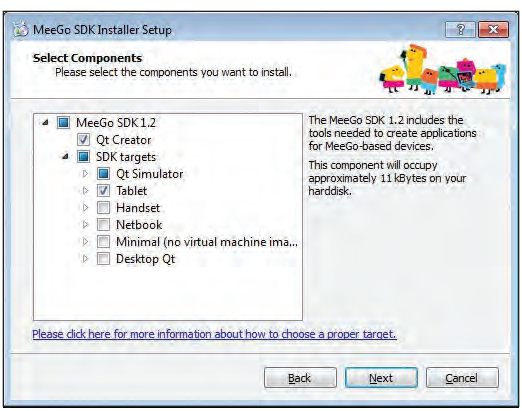
Getting started: If you want to develop apps for MeeGo, then you need to set the environment. In this case, you need to install the SDK

Install MeeGo Target and Runtime

Here, you select what components and SDK targets you want to install. These SDK targets are around 2GB and you will need one for most purposes. Have a look at these descriptions before installing.

Tools: These form the core of the MeeGo SDK

Targets: A target is a profile, such as "handset" or "tablet".This consists of 3 components: toolchain, build libraries and the virtual image machine with device skin. There are targets for x86 and ARM-based devices like handsets, netbooks and tablets. You must have at least one of these targets installed. Advanced users can opt for more than one.



Check: Select the components you want to install

On Windows, you will need either Microsoft Visual C++ compiler or MinGW compiler.

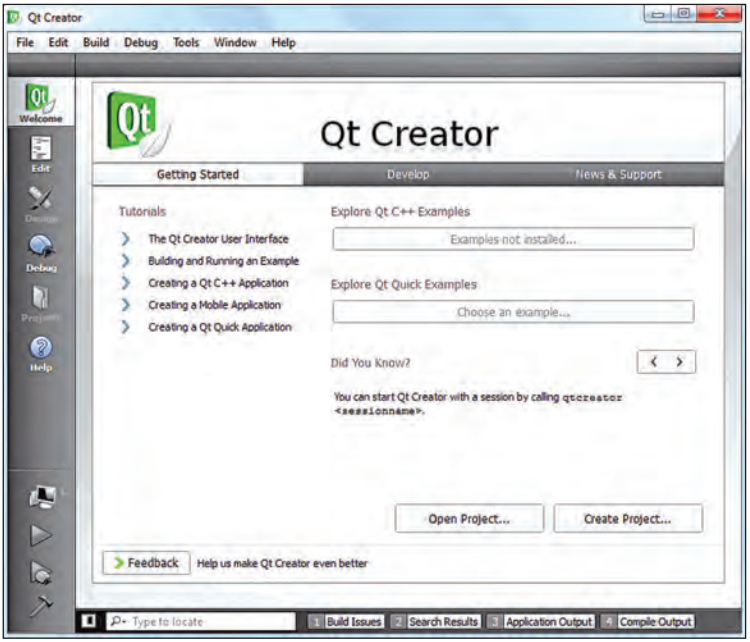
COMPONENT	DESCRIPTION
Qt Creator (143 MB)	This is a cross-platform Qt IDE.
Qt Simulator (800 MB)	Simulation Environment for Qt and Qt Mobility
QEMU Virtual Machine (11 MB)	This boots the MeeGO image on your host system

After the installation is complete you can add or remove SDK targets by going to "Maintain MeeGo SDK" tool in the MeeGo SDK group of Start menu

COMPONENT	DESCRIPTION
Toolchain	MeeGo-compatible compiler and tools
Build libraries	The MeeGo-compatible build libraries for compiling MeeGo applications
Virtual machine image	This MeeGo image will be used in the virtual machine

Configure Qt Creator

- Launch Qt Creator with Start > All Programs > MeeGo SDK > QtCreator.
- Configure Qt Creator to support MeeGo sysroot. Inside Qt Creator, follow these steps:
  - From the Tools menu, choose Options.
  - Click into the Qt4 section, and make sure the Qt Versions tab is open.
  - Click the + button, and then fill the Version name and Assign a "version name" (such as "meego tablet") and the "qmake location" of the MeeGo target.
    - Version name: meego-handset
    - qmake location: <MEEGO_SDK>\MADDE\targets\<target name>\bin\qmake.exe
      Replace <MEEGO_SDK> with the directory where you installed MeeGo SDK; the default is C:\MeeGoSDK_1.2; replace <target_name> with the name of the target (for example, meego-handset-ia32-w32-1.1).
    - Once added, select the new entry and click on the Rebuild button (bottom right) to build the debugging helper for this Qt version. This enables you to use debugging from Qt Creator.



New project: Launch your new project in Qt4

- Once you have created a new project, on the left side bar, click on the Project tab and click "Run" in the MeeGo target window, and select the configuration you just created. It will be named after the version name you gave in the previous step.

The result should look like this:

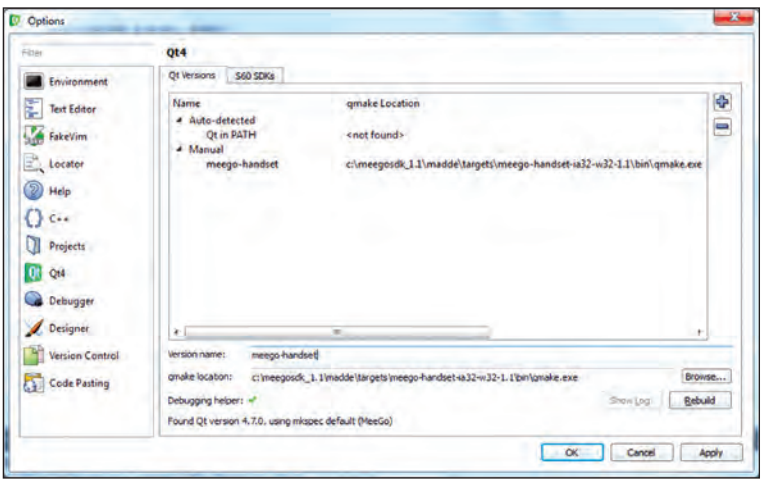
Setup the Virtual Machine (QEMU)

- Install kqemu accelerator for QEMU on 32-bit Windows system

Note: kqemu will only work on 32-bit Windows. 64-bit Windows will not work with kqemu. On 64-bit Windows systems, you can still build MeeGo applications, but you will need to run and debug them on a real device, instead of in a virtual machine (recommended approach, in any case, if you can obtain hardware).

- Download package from *http://wiki.qemu.org/download/kqemu-1.4.0pre1.tar.gz*
- Get the kqemu.inf and kqemu.sys files from above package.
- Depending on your version of Windows, you need to do the following –

- For Windows XP system, right click the kqemu.inf file and select "install".
- For Windows 7 or Windows Vista system, edit the kqemu.inf file with the changes below, and then right click the kqemu.inf and select "install".
  - [DefaultInstall.NT] --> [DefaultInstall]
  - [DefaultInstall.NT.Services] --> [DefaultInstall.Services]
  - [Uninstall.NT] --> [Uninstall]
  - [Uninstall.NT.Services] --> [Uninstall.Services]
- Start the kqemu manually. Select Start > All Programs > Accessories, click right mouse button over Command Prompt, and select "Run as administrator".



In the project setup dialog choose meego-handset

- In Command Prompt, run the following command:

```
> net start kqemu
```

- You need to start kqemu again after you reboot the system.

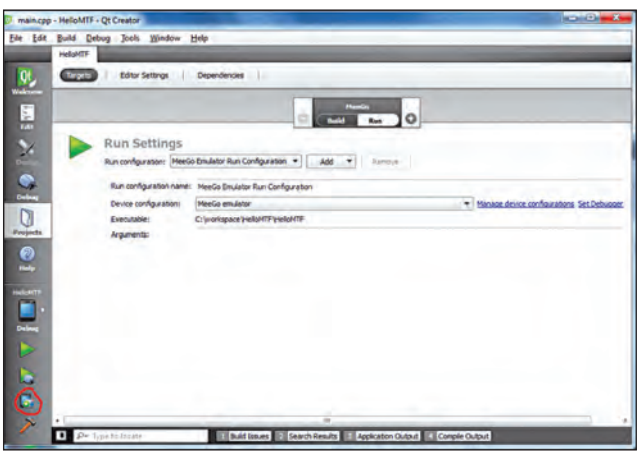
Use Qt Creator to develop MeeGo Applications

- Create a MeeGo touch project (for handset) or Qt Gui project (for netbook).
- Go to the menu: File > New File or Project > Qt Application Project > Meego Touch Framework Application.
- In "Project setup" dialog, choose "meego-handset".

- Go to the Project options in Qt Creator: Tools > Options > Projects > MeeGo Device Configurations.
- Click the "Add" button to create an emulator configuration, which will use QEMU or hardware device.
- To use QEMU, set the following values:
  - Device type: MeeGo emulator
  - Authentication type: Password
  - hostname: localhost (or 127.0.0.1)
  - SSH port: 6666
  - Username: root
  - Password: meego
- Click "Projects" again on the left side.
- Select "Run" in the MeeGo Build/Run box at the top. Make sure that the "Device configuration" is set to "MeeGo Emulator". This will ensure that QEMU is started when you run your application.
- You can now launch the QEMU MeeGo emulator. Click the "Start Meego emulator" icon near the bottom left corner of Qt Creator (circled in red in the screenshot below).

Note: You can also start the Qemu from MADDE terminal. For that you need to go to Start > Programs > MeeGoSDK > MADDE Terminal. In the terminal type:

- `mad list`
- `to see the runtime and target names, and then type:`
- `mad remote -r <runtime> poweron`



Launch the QEMU MeeGo emulator by clicking on the Start MeeGo emulator icon on the bottom right of the screen

- After QEMU starts, click the "Run" icon to run your application. Qt Creator will package and deploy your application to QEMU and start it on the target. You can see the window of your application in QEMU.
- You can also debug the application by clicking the "Debug" icon. Setting breakpoints and stepping are the same as local applications.

Configure Qt Creator to work with real devices

- To use a real device be sure that it is plugged into the network and that your firewalls are not blocking port 22, the SSH port.
- Install the mad-developer tools on the remote device.
- Go to the Project options in `Tools > Options > Projects > MeeGo Device Configurations` and click "Add" button. Click the "Add" button to create an emulator configuration, which will use QEMU or hardware device.
  - Device type: Remote device
  - Authentication type: Password
  - hostname: IP address of your hardware device running MeeGo
  - SSH port: 22
  - Username: root
  - Password: meego
- Open your project and in the "Projects" mode, click "Run" in the little MeeGo target window, and select the configuration you just created in "Device Configuration"
- Launch terminal inside the device, and run command "xhost +" in it.
- Click "Run" or "Debug" on the lower left corner of Qt Creator to run or debug your application on the configured Device.

INTRODUCTION TO QT

Qt is pronounced “cute”, is widely used as a cross platform framework in applications such as Google Earth.

Once you have installed Qt Creator, follow the steps in the following pages to get started...

- Click `Start > All Programs > MeeGo 1.1 SDK > Qt Creator.`
- Select Create Project
- Select Empty Qt project
- Select the directory you want to create the project in.
- And you are done!

Your first Qt Program - Hello Digit

- When you create a new Qt project, you get the following screen
- Right click on hello and add new C++ source file
- Type this simple code

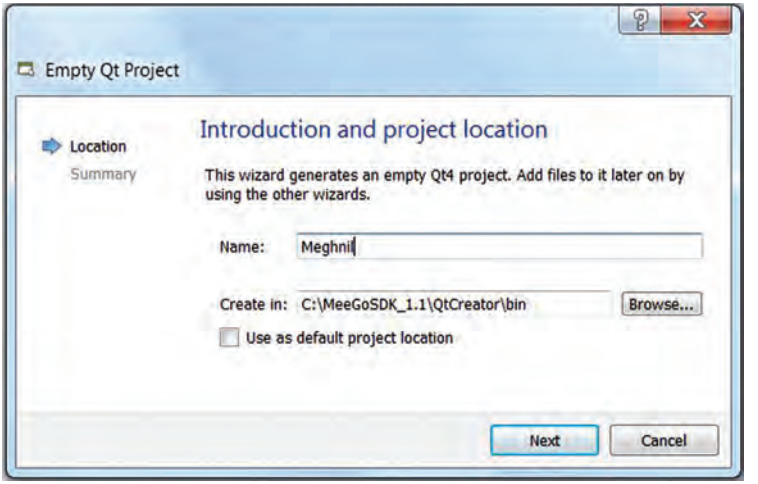
```
1 #include <QApplication>
2 #include <QLabel>
3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     QLabel *label = new QLabel("Hello Qt!");
7     label->show();
8     return app.exec();
9 }
```

Interpretation:

The first two lines include the definitions of the QApplication and QLabel classes. For every Qt class, there is a header file with the same name (and capitalization) as the class that contains the class's definition.

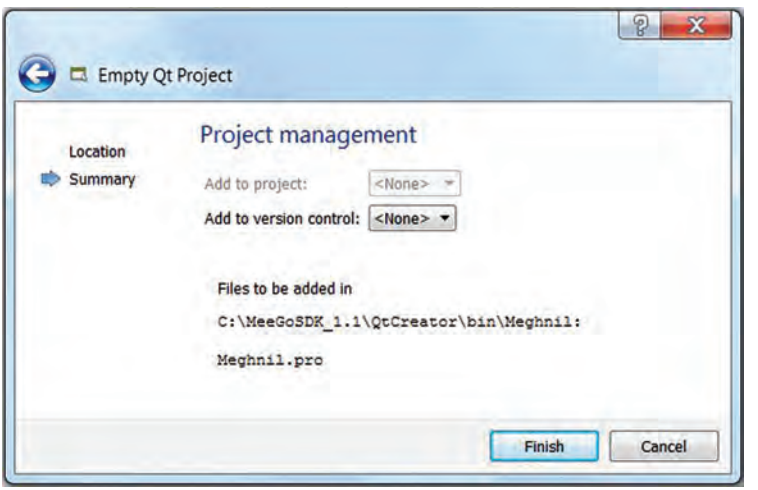
The fifth line creates a QApplication object to manage application-wide resources. The QApplication constructor requires argc and argv because Qt supports a few command-line arguments of its own.

The next line creates a QLabel widget that displays "Hello Qt!". In Qt and Unix terminology, a widget is a visual element in a user interface. The term stems from "window gadget" and is the equivalent of both "control" and "container" in Windows terminology. Buttons, menus, scroll bars, and frames are all examples of widgets. Widgets can contain other widgets; for example, an application window is usually a widget that contains a QMenuBar, a few QToolBars, a QStatusBar, and some other widgets. Most applications use a QMainWindow or a QDialog as the application window, but Qt is so flexible that any widget can be a window. In this example, the



Enter your details when you begin your project in Qt

QLabel widget is the application window.



You can gain from the added advantage of version control too

The next line makes the label visible. Widgets are always created hidden so that we can customize them before showing them, thereby avoiding flicker.

The last line passes control of the application on to Qt. At this point, the program enters the event loop. This is a kind of stand-by mode where the program waits for user actions such as mouse clicks and key presses. User actions generate events (also called "messages") to which the program can respond, usually by executing one or more functions. For example, when the user clicks a widget, a "mouse press" and a "mouse release" event are generated. In this respect, GUI applications differ drastically from conventional batch programs, which typically process input, produce results, and terminate without human intervention.

Output in Windows 7

Playing around with the code

To get a better understanding of how this works, let's try to tweak some of the lines

- `Replace the line`
- `QLabel *label = new QLabel("Hello Digit!");`
-
- `with`
- `QLabel *label = new QLabel("<h3><b>Hello</b> "`
- `"<font color=blue><i>Digit!</i></font></h3>");`
-
- `and rebuild the application.`
- `The tweaked output`

Connections

Now that we know how to get an output, we will now look at how to respond to user actions. The application consists of a button that the user can click to quit. The source code is very similar to "Hello Digit!", except that we are using a QPushButton instead of a QLabel as our main widget, and we are connecting a user action (clicking a button) to a piece of code.

- `#include <QApplication>`
- `#include <QPushButton>`
- `int main(int argc, char *argv[])`

```
{
    QApplication app(argc, argv);
    QPushButton *button = new QPushButton("Bye");
    QObject::connect(button, SIGNAL(clicked()),
                     &app, SLOT(quit()));
    button->show();
    return app.exec();
}
```

Let's analyse how this code is different form Hello Digit! Qt's widgets emit signals to indicate that a user action or a change of state has occurred. For instance, QPushButton emits a clicked() signal when the user clicks the button. A signal can be connected to a function (called a slot in that context) so that when the signal is emitted, the slot is automatically executed. In our example, we connect the button's clicked() signal to the QApplication object's quit() slot. The SIGNAL() and SLOT() macros are part of the syntax.

The Output

Simple Widget Layout

In this section, we will create a small example application that demonstrates how to use layouts to manage the geometry of widgets in a window and how to use signals and slots to synchronize two widgets. The application asks for the user's birth year which the user can enter by manipulating either a spin box or a slider.

Code

```
#include <QApplication>
#include <QHBoxLayout>
#include <QSlider>
#include <QSpinBox>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget *window = new QWidget;
    window->setWindowTitle("Enter your birth year");
    QSpinBox *spinBox = new QSpinBox;
    QSlider *slider = new QSlider(Qt::Horizontal);
    spinBox->setRange(1900, 2000);
    slider->setRange(1900, 2000);
```

Hello!: Here's your first application in MeeGo. You can continue modifying it using the steps mentioned here

```
    QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                     slider, SLOT(setValue(int)));
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                     spinBox, SLOT(setValue(int)));
    spinBox->setValue(1947);
    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(spinBox);
    layout->addWidget(slider);
    window->setLayout(layout);
    window->show();
    return app.exec();
}
```

Interpretation

The application consists of three widgets:

- QSpinBox,
- QSlider
- QWidget.

The QWidget is the application's main window. The QSpinBox and the QSlider are rendered inside the QWidget; they are children of the QWidget. The QWidget has no parent itself because it is being used as a top-level window. The constructors for QWidget and all of its subclasses take a QWidget * parameter that specifies the parent widget.

The QWidget will serve as the application's main window. We call setWindowTitle() to set the text displayed in the window's title bar.

The two QObject::connect() calls ensure that the spin box and the slider are synchronized so that they always show the same value. Whenever the

value of one widget changes, its valueChanged(int) signal -is emitted, and the setValue(int) slot of the other widget is called with the new value.

spinBox->setValue(1947) sets the spin box value to 1947. When this happens, the QSpinBox emits the valueChanged(int) signal with an int argument of 1947. This argument is passed to the QSlider's setValue(int) slot, which sets the slider value to 1947. The slider then emits the valueChanged(int) signal because its own value changed, triggering the spin box's setValue(int) slot. But at this point, setValue(int) doesn't emit any signal, since the spin box value is already 1947. This prevents infinite recursion. In short, changing one widget's value changes another.

One interesting thing to note here is that even though we didn't set the position or size of any widget explicitly, the QSpinBox and QSlider appear nicely laid out side by side. This is because QHBoxLayout automatically assigns reasonable positions and sizes to the widgets for which it is responsible, based on their needs.

Qt's approach to building user interfaces is simple to understand and very flexible. The most common pattern that Qt programmers use is to instantiate the required widgets and then set their properties as necessary. Programmers add the widgets to layouts, which automatically take care of sizing and positioning.

Simple QDialog

Now that we are done with the basics, let's create a QDialog.

Most GUI applications consist of a main window with a menu bar and toolbar, along with dozens of dialogs that complement the main window. It is also possible to create dialog applications that respond directly to the user's choices by performing the appropriate actions (e.g., a calculator application).

We will create our first dialog purely by writing code to show how it is done. Then we will see how to build dialogs using Qt Creator, Qt's visual design tool. Using Qt Creator is a lot faster than hand-coding and makes it easy to test different designs and to change designs later.

The Find Dialog

The source code is spread across two files: finddialog.h and finddialog.cpp.

finddialog.h.

```
#ifndef FINDDIALOG_H
```

```
#define FINDDIALOG_H
#include <QDialog>
class QCheckBox;
class QLabel;
class QLineEdit;
class QPushButton;
class FindDialog : public QDialog
{
    Q_OBJECT
public:
    FindDialog(QWidget *parent = 0);
signals:
    void findNext(const QString &str, Qt::CaseSensitivity
cs);
    void findPrevious(const QString &str,
Qt::CaseSensitivity cs);
private slots:
    void findClicked();
    void enableFindButton(const QString &text);
private:
    QLabel *label;
    QLineEdit *lineEdit;
    QCheckBox *caseCheckBox;
    QCheckBox *backwardCheckBox;
    QPushButton *findButton;
    QPushButton *closeButton;
};
#endif
```

Interpretation

#include <QDialog> includes the definition of QDialog, the base class for dialogs in Qt. QDialog is derived from QWidget.

The next two lines are forward declarations of the Qt classes that we will use to implement the dialog. A forward declaration tells the C++ compiler that a class exists, without giving all the detail that a class definition (usually located in a header file of its own) provides.

The Q_OBJECT macro at the beginning of the class definition is necessary for all classes that define signals or slots.

The FindDialog constructor is typical of Qt widget classes. The parent parameter specifies the parent widget. The default is a null pointer, meaning that the dialog has no parent.

The signals section declares two signals that the dialog emits when the user clicks the Find button. If the Search backward option is enabled, the dialog emits findPrevious(); otherwise, it emits findNext().

The signals keyword is actually a macro. The C++ preprocessor converts it into standard C++ before the compiler sees it. Qt::CaseSensitivity is an enum type that can take the values Qt::CaseSensitive and Qt::CaseInsensitive.

In the class's private section, we declare two slots. To implement the slots, we will need to access most of the dialog's child widgets, so we keep pointers to them as well. The slots keyword is, like signals, a macro that expands into a construct that the C++ compiler can digest.

For the private variables, we used forward declarations of their classes. This was possible because they are all pointers and we don't access them in the header file, so the compiler doesn't need the full class definitions. We could have included the relevant header files (<QCheckBox>, <QLabel>, etc.), but using forward declarations when it is possible makes compiling somewhat faster.

- `findiaalog.cpp`

We will now look at finddialog.cpp, which contains the implementation of the FindDialog class and we also be providing the interpretations side-by-side, since the code is quite long

- `1 #include <QtGui>`
- `2 #include "finddialog.h"`
- `3 FindDialog::FindDialog(QWidget *parent)`

First, we include <QtGui>, a header file that contains the definition of Qt's GUI classes. Qt consists of several modules, each of which lives in its own library. The most important modules are QtCore, QtGui, QtNetwork, QtOpenGL, QtScript, QtSql, QtSvg, and QtXml. The <QtGui> header file contains the definition of all the classes that are part of the QtCore and QtGui modules. Including this header saves us the bother of including every class individually.

In finddialog.h, instead of including <QDialog> and using forward declarations for QCheckBox, QLabel, QLineEdit, and QPushButton, we

could simply have included <QtGui>. However, it is generally bad style to include such a big header file from another header file, especially in larger applications.

```
4     : QDialog(parent)
5 {
6     label = new QLabel(tr("Find &what:"));
7     lineEdit = new QLineEdit;
8     label->setBuddy(lineEdit);
9     caseCheckBox = new QCheckBox(tr("Match &case"));
10     backwardCheckBox = new QCheckBox(tr("Search &back-
ward"));
11     findButton = new QPushButton(tr("&Find"));
12     findButton->setDefault(true);
13     findButton->setEnabled(false);
14     closeButton = new QPushButton(tr("Close"));
```

On line 4, we pass on the parent parameter to the base class constructor. Then we create the child widgets. The tr() function calls around the string literals mark them for translation to other languages. The function is declared in QObject and every subclass that contains the Q_OBJECT macro. It's a good habit to surround user-visible strings with tr(), even if you don't have immediate plans for translating your applications to other languages.

In the string literals, we use ampersands ('&') to indicate shortcut keys. For example, line 11 creates a Find button, which the user can activate by pressing Alt+F on platforms that support shortcut keys. Ampersands can also be used to control focus: On line 6 we create a label with a shortcut key (Alt+W), and on line 8 we set the label's buddy to be the line editor. A buddy is a widget that accepts the focus when the label's shortcut key is pressed. So when the user presses Alt+W (the label's shortcut), the focus goes to the line editor (the label's buddy).

On line 12, we make the Find button the dialog's default button by calling setDefault(true). The default button is the button that is pressed when the user hits Enter. On line 13, we disable the Find button. When a widget is disabled, it is usually shown grayed out and will not respond to user interaction.

```
15   connect(lineEdit, SIGNAL(textChanged(const QString
&)),
16           this, SLOT(enableFindButton(const QString
&)));
```

```
17    connect(findButton, SIGNAL(clicked()),
18            this, SLOT(findClicked()));
19    connect(closeButton, SIGNAL(clicked()),
20            this, SLOT(close()));
```

The private slot enableFindButton(const QString &) is called whenever the text in the line editor changes. The private slot findClicked() is called when the user clicks the Find button. The dialog closes itself when the user clicks Close. The close() slot is inherited from QWidget, and its default behavior is to hide the widget from view (without deleting it). We will look at the code for the enableFindButton() and findClicked() slots later on.

Since QObject is one of FindDialog's ancestors, we can omit the QObject:: prefix in front of the connect() calls.

```
21    QHBoxLayout *topLeftLayout = new QHBoxLayout;
22    topLeftLayout->addWidget(label);
23    topLeftLayout->addWidget(lineEdit);
24    QVBoxLayout *leftLayout = new QVBoxLayout;
25    leftLayout->addLayout(topLeftLayout);
26    leftLayout->addWidget(caseCheckBox);
27    leftLayout->addWidget(backwardCheckBox);
28    QVBoxLayout *rightLayout = new QVBoxLayout;
29    rightLayout->addWidget(findButton);
30    rightLayout->addWidget(closeButton);
31    rightLayout->addStretch();
32    QHBoxLayout *mainLayout = new QHBoxLayout;
33    mainLayout->addLayout(leftLayout);
34    mainLayout->addLayout(rightLayout);
35    setLayout(mainLayout);
```

Next, we lay out the child widgets using layout managers. Layouts can contain both widgets and other layouts. By nesting QHBoxLayouts, QVBoxLayouts, and QGridLayouts in various combinations, it is possible to build very sophisticated dialogs.

The Find dialog's layouts

One subtle aspect of the layout manager classes is that they are not widgets. Instead, they are derived from QLayout, which in turn is derived from QObject. In the figure, widgets are represented by solid outlines and layouts are represented by dashed outlines to highlight the difference between them. In a running application, layouts are invisible.

When the sublayouts are added to the parent layout (lines 25, 33, and 34), the sublayouts are automatically reparented. Then, when the main layout is installed on the dialog (line 35), it becomes a child of the dialog, and all the widgets in the layouts are reparented to become children of the dialog.

```
36     setWindowTitle(tr("Find"));
37     setFixedHeight(sizeHint().height());
38 }
```

Finally, we set the title to be shown in the dialog's title bar and we set the window to have a fixed height, since there aren't any widgets in the dialog that can meaningfully occupy any extra vertical space. The QWidget::sizeHint() function returns a widget's "ideal" size.

This completes the review of FindDialog's constructor. Since we used new to create the dialog's widgets and layouts, it would seem that we need to write a destructor that calls delete on each widget and layout we created. But this isn't necessary, since Qt automatically deletes child objects when the parent is destroyed, and the child widgets and layouts are all descendants of the FindDialog.

Now we will look at the dialog's slots:

```
39 void FindDialog::findClicked()
40 {
41     QString text = lineEdit->text();
42     Qt::CaseSensitivity cs =
43             caseCheckBox->isChecked() ?
Qt::CaseSensitive
44                                       :
Qt::CaseInsensitive;
45     if (backwardCheckBox->isChecked()) {
46         emit findPrevious(text, cs);
47     } else {
48         emit findNext(text, cs);
49     }
50 }
```

```
51 void FindDialog::enableFindButton(const QString &text)
52 {
53     findButton->setEnabled(!text.isEmpty());
54 }
```

The findClicked() slot is called when the user clicks the Find button. It emits the findPrevious() or the findNext() signal, depending on the Search backward option. The emit keyword is specific to Qt; like other Qt extensions it is converted into standard C++ by the C++ preprocessor.

The enableFindButton() slot is called whenever the user changes the text in the line editor. It enables the button if there is some text in the editor, and disables it otherwise. These two slots complete the dialog.

Testing

We can now create a main.cpp file to test our FindDialog widget:

```
1 #include <QApplication>

2 #include "finddialog.h"

3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     FindDialog *dialog = new FindDialog;
7     dialog->show();
8     return app.exec();
9 }
```

WORKING WITH QT CREATOR

Qt Creator is a complete integrated development environment (IDE) for creating applications with Qt Quick and the Qt application framework.

Qt Creator has been designed with the following things in mind: Develop applications and user interfaces and deploy them across several desktop and mobile operating systems. Allow a team of developers to share a project across different development platforms (Microsoft Windows®, Mac OS X®, and Linux®) with a common tool for development and debugging. Allows UI designers to join the team, by providing them with Qt Quick tools for creating fluid user interfaces in close cooperation with the developers. Meet the development needs of Qt Quick developers who are looking for simplicity, usability, productivity, extendibility and openness, while aiming to lower the barrier of entry for newcomers to Qt Quick and Qt.

Working with Qt Creator

Qt Creator has developed a concept of "Modes" to meets its design goals of simplicity, ease-of-use, and productivity. These modes adapt the user interface to the different application development tasks at hand. Each mode has its own view that shows only the information required for performing a given task, and provides only the most relevant features and functions related to it. You can employ the mode selector to switch to a Qt Creator mode.

Creating projects

In order to build and run applications in Qt Creator, the required information is specified in the project build and run settings.

Setting up a new project in Qt Creator is aided by a wizard that guides the user step-by-step through the project creation process.

- Select the type of project from the available categories.
- When creating Qt Quick Projects, the user can select either:
  a. Qt Quick UI
     A Qt Quick UI project contains a single QML file that defines the main view of the application. UI designers can use it to create an application user interface and review it in the QML Viewer, without having to build the application.
  b. Qt Quick Application.
     Developers can build Qt Quick applications and deploy them on mobile target platforms.
- A Qt Quick UI project can be easily converted into a Qt Quick application by using the Qt Quick application wizard to import the main QML file in the Qt Quick UI project.
- Enter the settings needed for a particular type of project, when prompted.
- Qt Creator automatically generates the project with required headers, source files, user interface descriptions and project files, as defined by the wizard.

Not only does the wizard help new users get up and running quickly, it also enables more experienced users to streamline their workflow for the creation of new projects. The Qt Quick application wizard allows developers to create projects that they can deploy on mobile devices with a click of the run button.

Designing user interfaces

Qt Creator 2.1 allows both UI designers and developers to work together without any compromises. Both are free to use their preferred tools:

UI designers can work in Photoshop or Gimp and use a QML export script to export their designs to Qt Creator.

Developers can then add the necessary code to complete the application.

If more changes are needed, UI designers can make them in Qt Quick Designer. It is also possible to design the user interface from start to finish in the Qt Quick Designer.

The Qt quick designer

The Qt Quick Designer interface provides the following features:

- The center of the Qt Quick Designer view is used for the construction of the user interface.
- The available building blocks (items and resources) are kept in the Library on the left side of the window.
- Reusable elements that you copy to the project folder are automatically added to the Library.
- The Navigator displays the QML elements in the current QML file;
- The Properties organizes the properties of the selected QML element or QML component.
- The State displays the different states of the component.

Coding

The code editor is one of the key components of Qt Creator, which allows you to write, edit and navigate in source code. The code editor can be used in the Edit mode to write QML code.

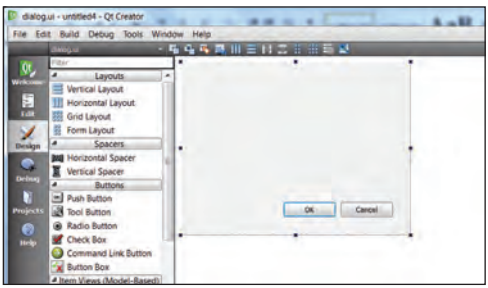
The code editor offers a number of features :

- Code completion for elements, properties, ids and code snippets.
- Checking code syntax and marking errors (with wavy underlining in red) while editing, to find typos and syntax errors.
- Qt Quick Toolbars for specifying properties of QML elements.
- Syntax highlighting for keywords, symbols, and macros in QML files.
- Incremental search that highlights the matching strings in the window while typing.
- Advanced search that allow developers to search from currently open projects or files on the file system.

The code editor supports different keyboard shortcuts for faster editing. It is possible to work without using the mouse at all, allowing developers to keep their hands on the keyboard and work more productively.

Completing Code

As developers write code, Qt Creator suggests properties, IDs, and code snippets to complete the code. It provides a list of context-sensitive suggestions to the statement currently under the cursor.



Dialog UI: This is where you set the menu of your application

Code snippets can consist of multiple fields that developers specify values for. Developers can select an item in the list and press Tab or Enter to complete the code.

Applying Refactoring Actions

Qt Creator allows developers to quickly and conveniently apply actions to refactor code by selecting them in a context menu. The actions available depend on the position of the cursor in the code editor.

To apply refactoring actions, developers can select an element, and then select an action in the context menu. You can -

- Rename IDs to give items a meaningful ID and update all references.
- Split initializers to place each property on a separate line.
- Move a QML element into a separate file to reuse it in other .qml files.

Using Qt Quick Toolbars

When users edit QML code in the code editor, it becomes difficult to specify the properties of QML components. For example, few people can visualize the color #18793f.

To easily edit these properties, users can employ the Qt Quick Toolbars. When a component is selected in the code and a toolbar is available, a light

bulb icon appears. You can then select the icon to open the toolbar. Qt Quick Toolbars are available for editing the properties of the following QML elements: Rectangles, Text, Images and Animation

Debugging

Qt Creator comes with a Debug mode. It allows you to inspect the state of the application while debugging JavaScript functions. You can set breakpoints, view call stack trace, and examine locals and watchers.

Suppose, your application is interrupted by a breakpoint; you can then use the QML Script Console to execute JavaScript expressions in the current context.

You can even type JavaScript expressions and use them to get information about the state of the application, such as property values. If you change property values or add properties in the code editor, the changes are automatically updated.

While the application is running, you can use the QML Observer view to explore the object structure, debug animations, and inspect colors. You can even use the "Observe mode" to jump to the position in code where an element is defined.

Deploying Applications to Mobile Devices

Qt Creator deploy configurations handle the packaging of the application as an executable and copying it to a location developers want to run the executable at. The files can be copied to a location in the file system of the development PC or a mobile device. To deploy files on mobile devices, you can either connect the devices to the development PC or use the installation packages generated by Qt Creator.

Note that you will have to convert Qt Quick UI projects into Qt Quick applications for deployment on mobile devices.

Getting Help

All the Qt documentation and examples are accessible via the Qt Help plugin in Qt Creator. To view the documentation, the "Help Mode" is used.

If you are in the "Edit mode" you can access context sensitive help by moving the text cursor to a Qt class or function and then press the F1 key. The documentation is displayed within a panel on the right side of the code editor, as shown in the following figure.

Summary

Qt Creator offers a holistic development environment for the successful creation of Qt Quick applications. It is a a very light tool and it focuses on the needs of Qt Quick developers and UI designers, productivity, and usability.

The Qt Creator mode-centric way of working helps to keep the focus on the task at hand. It is equally appealing to both UI Designers and App Developers. Qt Quick Designer, Qt Quick Toolbars, Code editor, QML Observer make Qt Creator the ideal environment for developing Qt Quick applications. Support for cross-platform build systems and version control software ensures that Qt Creator can be integrated fully into the working environment of a development team.

HOW TO CREATE UI LAYOUTS

The interface of your application would decide whether users find it intuitive and easy to use. Here are a few guidelines to creating UI layouts in MeeGo.

Overall UI Model

The Handset UI for MeeGo consists of the following main components:

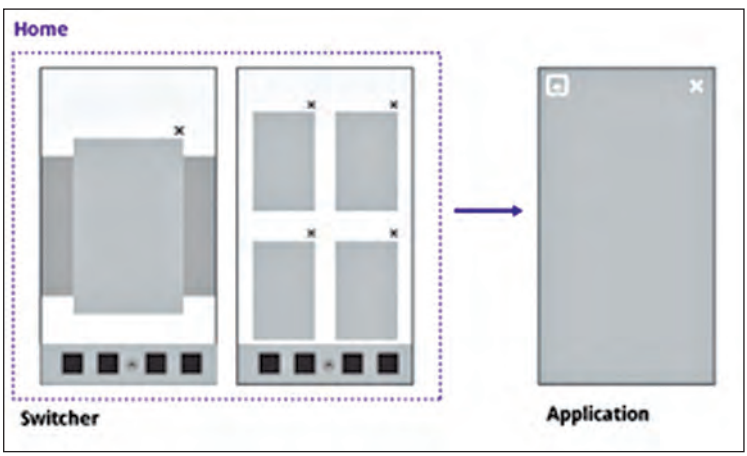
- Lock Screen: The lock screen is presented when the user presses the power key to wake the device from idle state.
- Home: It provides access to all the open applications through the switcher. You can access the launcher as well as your favorite apps.
- Launcher: The launcher is a view that contains links to all applications installed in the device. In the launcher, the user can browse through the applications and add up to 4 links to the quick launch bar at the bottom of the screen. Applications are presented in a 4 x 4 grid. In the case where there are more than 16 applications, more pages of the same

grid are added to the right. Paging through the different grids is done by swiping the current grid off screen, hence bringing the new one into view.

- Switcher

You can multitask on MeeGo by using the area called Switcher. You can open multiple applications at once, such as listening to music while writing a message. The basic behavior of the switcher is as follows:

When an application is first launched, a new task starts, presented by a live thumbnail. If you open a second application the new task is included within the switcher. You can now quickly switch between applications by pressing the home button. When the user clicks on an application in the launcher, that is already open in the switcher, the system opens the window that was already open.



Flow from home to the application window

How are my applications arranged?

The default view of the switcher presents the latest accessed task in focus, with the other, different tasks running to the right. Every time a new task is opened, it pushes the previous task panels to the left and adds its thumbnail view to the far right.

How do I Browse through my apps ?

You can browse through the thumbnails by:

- Dragging the thumbnail.
- Quickly swiping to pan from one end to the other..

How do I Change my Mode?

You can use a pinch multi-touch motion to change the switcher into an overview mode. By pinching in, thumbnails move into a grid display. Pinching out, while in the grid mode, changes the view back to larger stacked mode.

Core Interactions

UI Feedback

Direct feedback

It refers to the response the user receives when using the touch UI. It can be a haptic, auditory, or visual feedback from interaction with the touch screen or hardware buttons.

Indirect feedback

It refers to the response the user receives while, either not using the device, or using it so that the attention is not directed to the UI. Indirect feedback is typically in the form of notifications.

Notifications can be visual, auditory,tactile or haptic.

- When feedback is needed almost immediately after the user's last action (such as when the user tries to send an email without defining a recipient), a visual notification is usually used because the user's attention is already on the device.
- Auditory and haptic feedbacks are used to emphasize the visual message.
- Haptic feedback alone is usually used when you do not know whether the user's attention is on the device.
- Tactile feedback alone is usually used when it is appropriate to confirm the user's action, but a visual confirmation message is too much.

The presentation of alerts and notifications depends both on the settings in the active profile and on the current UI state. For example, during a phone call, the auditory part of many alerts is changed (presented as a beep, or the auditory part can be a simplified version of the original, or the sound may be omitted altogether).

Text Input

MeeGo provides support to both virtual and hardware keyboards.

The virtual keyboard is invoked automatically when an input field is in focus. It has a portrait and landscape layout, opening according to the orientation it is being called from. If the user rotates the device, the layout is smoothly changed according to the new orientation. The user can close

the virtual keyboard by either tapping outside the text field or by dragging down the virtual keyboard.

Full Screen Mode

MeeGo also provides a Full Screen Mode for applications that need to provide a more immersive experienceFull Screen can be presented in two different ways:

- Controls a tap away

  All controls (Title Bar, Status, extra controls, etc) are removed, presenting only the maximized content. Controls are made visible by tapping the screen.
- Fixed controls

  Since MeeGo also runs in devices with no other hardware navigation keys, it provides a persistent way out of the application like "back" or "close".

  Your home screen is usually divided into four parts:

Status Bar: It shows signal strength, time, and battery life and notifications.
Title Bar: It contains the homescreen button, title, or tabs, View and the close or back button.
Content Area: This is a dynamic area which presents navigation options, text, media, etc.
Tool Bar/Tab Bar: This is an optional fixed bar available to an application at all times, whenever core commands are needed. The bar is limited to 4 actions

Customisation available to you as a User

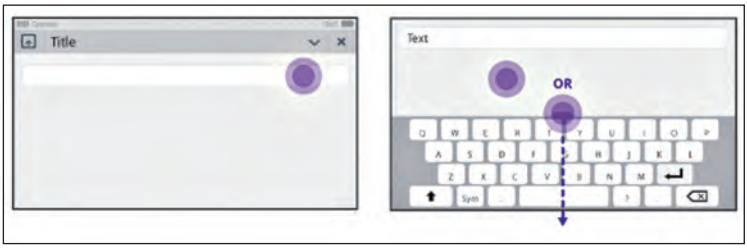
Portrait vs. Landscape

Different applications require different device orientation viz. Landscape or Potrait. You might want to watch a video in the Landscape view but scroll lists in portrait view.

MeeGo common components are always provided for both orientations. Applications can overwrite default changes and the assets can be subtly resized and/or reshuffled to make better use of the screen real estate after changing the orientation.

1 vs. 2 Columns

Lists are probably the most common UI layout, and have a particular behavior depending on the orientation. The portrait allows for more items to be presented while landscape view presents the possibility to extend the content area of a particular item. Some applications may consider splitting lists into two columns to optimize space usage.



A tap on the search bar should draw out an input tool, a QWERTY in this case

Settings

The Settings view presents Home, Title, and Back. In the Settings view group similar settings are grouped together to help the user find related options.

Whenever you make any changes in the settings, the change takes place immediately, without the need to save those changes. You can use the Undo button in the view menu to discard any changes, and reset all changed settings back to the previous states.

Navigation within your applications

MeeGo supports three main templates to help support the application's main functional goals.

Drill Down

It is used when access to all application navigation structure is needed on the top levelThe Close button gets replaced by a Back button, as you drill down the structure.

Navigation in View Menu

It is used for flat experiences, prioritizing content over navigation structure. View Menu is used for hopping between navigational options. It allows for many navigational options, as opposed to the other templates. Filtering works like a drill down, hence the Close button gets replaced by a Back button, for example, switching between Messaging accounts.

Tab Bar

This template is for quick access between distinct areas or modes. This model is limited to maximum 4 navigation options which does not change over time.

Basic Visual Customisation

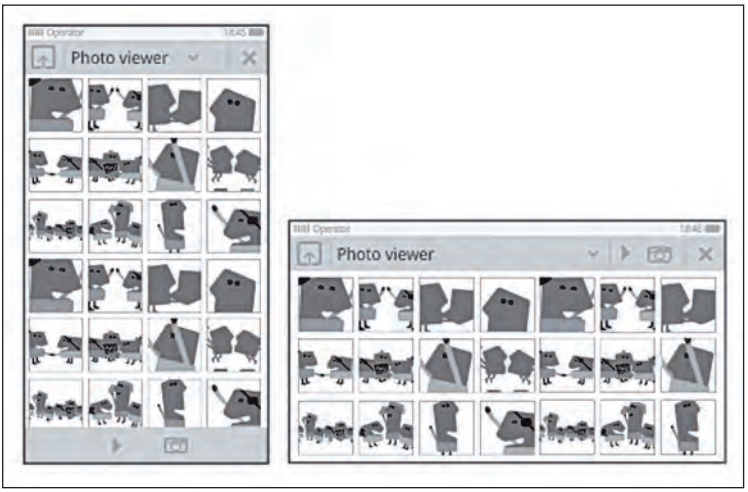
Colors

The base theme has three main colors:

- Normal state: Light grey.
- Pressed state: Dark grey.
- Selected state: Blue

Application background

You can also customise the background image, or, if it's not specified, the background color. The home and lock screen backgrounds are different.



Alternating between portrait and landscape views when viewing photos

How to customize

The application background is in the Images folder. It is called duiapplicationpage-background.png. The background file can be changed using Photoshop or any bitmap tool.

Dependencies

The background has to be carefully designed with all the other elements in the interface. The application background is what shows under all the buttons, switchers, and label texts.

Selected state gradient

You can quickly get the principal brand color into a theme by changing the selected state colors (like blue in the figure below) to the brand colors.

How to customise

- Take a backup of the svg folder.
- Open the svg file from svg folder with Inkscape or Illustrator CS4 and above.
- Modify the colored gradient to your brand colors.
- Save the graphic as svg tiny.
- Test the graphics in the device.
- Repeat steps 2-4 to all selected state graphics in the svg folder.

Dependencies

You will also have to update the text colors in the constants.ini file.If something doesn't work in the new theme, the software always reverts back to the base theme graphics.

Gradients of all button states

You can customize the gradients to change the look and feel of the buttons and other elements. You can give a different 3d form to a button , change the lighting effect and direction and even modify the feel of a texture. It can also be used for changing the color logic.

How to customize

The steps for customizing are the same as for changing the highlight color. The same logic has to be extended to all states of buttons, lists, switches, and other elements.

Dependencies

You will need to adjust the text and background colors.

INTRODUCTION TO QML

QML is a declarative language designed to describe the user interface of a program: both what it looks like, and how it behaves. In QML, a user interface is specified as a tree of objects with properties.

This introduction has been kept very simple, so even people with little or no programming experience can feel at ease. JavaScript is used as a scripting language in QML, so you may want to learn a bit more about it before diving deeper into QML. It's also helpful to have a basic understanding of other web technologies like HTML and CSS, but it's not mandatory.

Intro to Qt Quick

Overview

QML is a high level, scripted language. Its commands (known as elements) leverage the power and efficiency of the Qt libraries to make easy to use commands that perform intuitive functions. Draw a rectangle; display an image at a position and so on. Behind these elements are complex C++ libraries that efficiently perform the action.

The language also allows more flexibility of these commands by using Javascript rather than C++ to add new layers of logic to your application. Javascript is easier to learn than C++ and can be embedded into the QML files or imported from a separate file.

In QML the types of various 'objects' are referred to as elements.

An element usually has various properties that help define the element. For example, if we created an element called Circle then the radius of the circle would be a property.

A First Look

The basic syntax of an element is

```
SomeElement {
    id: myObject
    ... some other things here ...
}
```

Here we are defining a new object. We specify its 'type' first as SomeElement. Then within matching braces { ... } we specify the various parts of our element.

The id is a unique identifier for the element, it must start with a lower case letter and only contain letters, numbers and underscores. It is this particular object's name. If this SomeElement element was a Rectangle instead and it was one of many then the optional unique id would allow us to manipulate each element individually.Each visual element is ultimately based on, or inherits from, an element called Item. Item has certain properties and actions that may be useful.

Take a simple element such as a Rectangle. It has an id, we will call it myRectangle, it has a width and a height. Imagine that we want a rectangle that is 500 pixels by 400 pixels in the x and y directions (horizontal by vertical).

We can implement this Rectangle with these properties this way

```
import QtQuick 1.0

// This is a comment.

Rectangle {
    id: myRectangle
    width: 500
    height: 400
}
```

This is a valid QML script. To run it, copy it and save it to a file, say myexample.qml, and on the command line run the following command:

```
qmlviewer myexample.qml
```

On Mac OS X, open the "QMLViewer" application instead and open the myexample.qml file, or run it from the command line:

QMLViewer.app/Contents/MacOS/QMLViewer myexample.qml

It will create a very boring rectangle in its own window.

Hello Digit!

We can now add some color and text to make a Hello Digit QML program.

Rectangle has the property color to produce a background color.

Text is handled by a different element called Text. We need to create a Text object inside the Rectangle and set its text property to "Hello Digit!". So to set the text to "Hello Digit!" and the background colour to light gray,

```
import QtQuick 1.0

Rectangle {
    id: myRectangle
    width: 500
    height: 400

    Text { text: "Hello Digit!" }

    color: "blue"
}
```

From now on we will not always show the import statement for Qt but it should still be there when you create your QML scripts.

To make our Hello Digit example a little nicer set the position of the text to be at pixel position x = 100, y = 100 within the displayed window. This

position belongs to the Text element so we set the position inside its definition. Note that we separate different QML statements on the same line with a semi-colon, or we could have simply put each statement on a new line

```
Text {
    text: "<h2>Hello Digit</h2>"; color: "green"
    x: 100; y:100
}
```

Not only did we reposition the text, but the text was altered by adding HTML tags to change the font size. The text color was also changed from the default black to dark green by using a standard string for the color's SVG name.

We could also have used a hexadecimal string for the RGB (red-green-blue, as #rrggbb) values of the color similar to the method used in HTML,

```
Text {
    text: "<h1>Hello Digit</h1>"
    color: "#882200"
    x: 100; y: 100
}
```

All of these changes occurred within the Text object which is the scope of these property changes.

Other objects may use the information but it belongs to the element where the property has been defined.

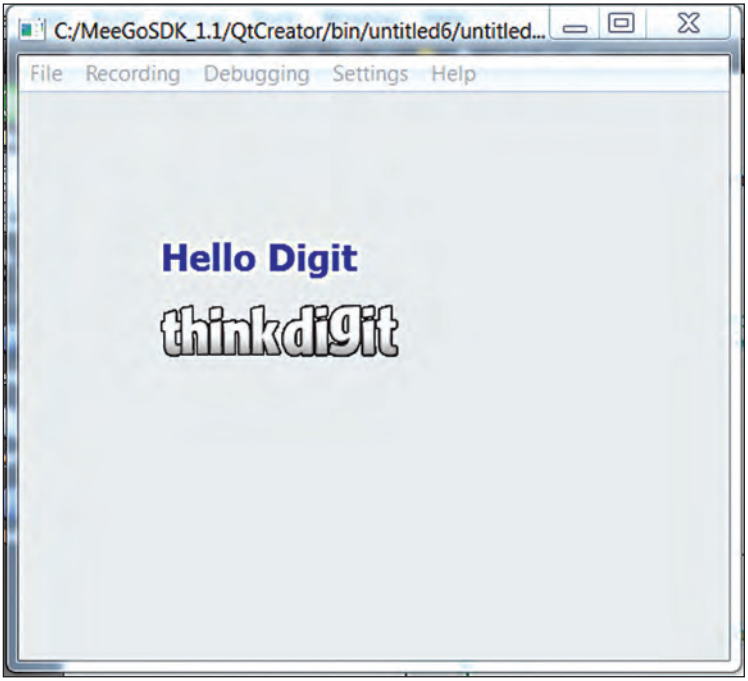
Images

To add an image to our little application we use the Image element. An Image uses a path to an image file, and has properties to control the aspect ratio, the image size, to tile the area amongst others. The source of the image, the path to the file, is a URL. Therefore the file can be local or it can be remote.

```
Image {
    source: "img/digit.gif"
}
```

This displays the image, as we would expect, at the top left of the window. The position of the default x = 0, y = 0 coordinate. The example here uses a PNG file, but it could have been one of various supported formats, including JPG and GIF.

Let us reposition the image and enlarge it. Place it at the same 'x' offset as the "Hello world again" text, but put it another 50 pixels below the text, also make it 150 by 150 pixels in size,



Getting the output was really simple

```
    Image {
        source: “img/digit.gif”
        x: 100; y: 150
        width: 167; height: 36
    }
```

Adding the Hello Digit example, with the text and the image example we can write a simple piece of QML that starts to look a bit better.

```
import QtQuick 1.0

Rectangle {
    id: myRectangle
    width: 500
    height: 400
    Text {
        text: “<h1>Hello Digit</h1>”
```

```
        color: "#002288"
        x: 100; y: 100
    }
    Image {
        source: "img/digit.gif"
        x: 100; y: 150
        width: 167; height: 36
    }
    color: "lightgray"
}
```

The result is still quite simple

Anchors: Aligning Elements

Using absolute positioning, such as saying x = 100 and y = 150, works well until the user or developer stretches or increases the size of the window. Then the positions need to be recalculated. What would be nice would be a relative means of positioning of objects in a window or rectangle. For example, we want to place an image at the bottom of a rectangle, we would like to specify the image's location as the 'bottom of the window', not a specific coordinate. We can do this with the anchors property, which objects inherit from Item.

The anchors property is really a property group. It is a collection of related properties. It has properties within it which can be used by means of the dot notation.

The dot notation uses object ids and property names to use a particular object or property. Say I have a rectangle r1, which contains a rectangle r2, which contains an Item item1, which has an 'x' property I want to change. I just use the dot notation to identify it: r1.r2.item1.x

If we want to position an image at the bottom of the rectangle it is inside. I have to specify that the bottom of the image is also at the bottom of the rectangle

```
import QtQuick 1.0

Rectangle {
    id: myWin
    width: 500
```

```
    height: 400
    Image {
        id: image1
        source: "img/digit.gif"
        width: 167; height: 36
        anchors.right: myWin.right
    }
}
```

This places the logo at the top right of the window.

If you're not satisfied with the outcome, you can move elements around

We would like it centered and not touching the bottom of the window, for aesthetic reasons. For the centering we use the horizontalCenter property, and to prevent the touching of the image to the bottom of the rectangle, the bottomMargin property is used. So the new actions for the script are

- set the bottom of the image (anchors.bottom) to be the bottom of the window
- move the image to be in the horizontal center of the window
- set a margin of 100 pixels so that the image does not touch the bottom window border

Encoded into QML the script becomes

```
import QtQuick 1.0

Rectangle {
    id: myWin
```

```
    width: 500
    height: 400
    Image {
        id: image1
        source: “img/digit.gif”
        width: 167; height: 36
        anchors.bottom: myWin.bottom
        anchors.horizontalCenter: myWin.horizontalCenter
        anchors.bottomMargin: 100
    }
}
```

Run this and resize the window. You will see that now the position of the image adjusts during the resize.

You can also add another object say a block of descriptive text and place it above or below the image or to the side. This code places some text just above the image

```
    Text {
        text: “<h2>ThinkDight</h2>”
        anchors.bottom: image1.top
        anchors.horizontalCenter: myWin.horizontalCenter
        anchors.bottomMargin: 15
    }
```

Please note that anchors is a property group, to be used within the object. When referencing these properties from another object we use the property directly, instead of saying:

```
myRectangle.anchors.top  // Wrong
```

we use

```
myRectangle.top          // Correct
```

Transformations

To get additional effects we can transform our image. We can, for instance rotate a piece of text by 180 degrees to display upside-down text or even rotate an image by 90 degrees to lay it on its side. These transformations require additional information.

For rotation, the additional information includes:

- The origin relative to the object being rotated.
- The axis of rotation
- The angle in degrees to rotate the image through in a clockwise direction.

We can rotate the text and even scale the text. The transform property is a list of Transform elements, so using the list syntax, we can produce a list of transformations.

```
myList: [ listElement1, listElement2, ... } ]
```

The text will be rotated by 180 degrees anti-clockwise and scaled vertically by a factor of 1.1 and by 1.3 horizontally.

Using the example above as the basis for this we have,

```
import QtQuick 1.0

Rectangle {
    id: myWin
    width: 500
    height: 400
    Image {
        id: image1
        source: “img/img.gif”
        width: 150; height: 150
        anchors.bottom: myWin.bottom
        anchors.horizontalCenter: myWin.horizontalCenter
        anchors.bottomMargin: 10
        transform: Rotation {
            origin.x: 75; origin.y: 75
            axis{ x: 0; y: 0; z:1 }  angle: -90
        }
    }
    Text {
        text: “<h2>Transformation</h2>”
        anchors.bottom: image1.top
        anchors.horizontalCenter: myWin.horizontalCenter
        anchors.bottomMargin: 15
        transform: [
            Scale { xScale: 1.1; yScale: 1.3 } ,
            Rotation {
                origin.x: 50; origin.y: 50
                axis{ x: 0; y: 0; z:1 }  angle: 180
            }
        ]
    }
```

```
}
```

The code block in image1 starting with transform specifies that the transform property will be a Rotation through -90 degrees, which is anti-clockwise, about the z-axis running through the center of the image at (75,75), since the image is 150 x 150 pixels.

The other transformation available is Translate. This produces a change in position of the item. Also note that the order of the transformations is important.

Animations

Animation in QML is done by animating properties of objects. In QML these are QML Basic Types named as real, int, color, rect, point, size, and vector3d. There are a number of different ways to do animation. Here we will look at a few of them.

Number Animation

Previously we have used a rotation transformation to change the orientation of an image. We could easily animate this rotation so that instead of a straight rotation counter-clockwise of 90 degrees we could rotate the image through a full 360 degrees in an animation. The axis of rotation won't change, the position of the center of the image will not change, only the angle will change. Therefore, a NumberAnimation of a rotation's angle should be enough for the task.

If we wish for a simple rotation about the center of the image then we can use the rotation property that is inherited from Item. The rotation property is a real number that specifies the angle in a clockwise direction for the rotation of the object.

Here is the code for our animated rotating image.

```
import QtQuick 1.0

Rectangle {
    id: mainRec
    width:  600
    height: 400
    Image {
```

```
        id: image1
        source: “img/digit.gif”
        x: 200; y: 100
        width: 167; height: 36
        // Animate a rotation
        transformOrigin: Item.Center
        NumberAnimation on rotation {
            from: 0; to: 360
            duration: 2000
            loops: Animation.Infinite
        }
    }
}
```

Also if we had used the Rotation transformation instead, then we would have more control over the various parameters. We could vary the axis, to be not just a different offset from the z-axis but along the y-axis, x-axis or combination. For example, if the task had been to animate the rotation about the y-axis passing through the center of the image then the following code would do it.

```
 import QtQuick 1.0

Rectangle {
    id: mainRec
    width:  600
    height: 400
    Image {
        id: image1
        source: “img/digit.gif”
        x: 200; y: 100
        width: 167; height: 36
        // Animate a rotation
        transform: Rotation {
            origin.x: 50; origin.y: 50; axis {x:0; y:1; z:0}
angle:0
            NumberAnimation on angle  {
                from: 0; to: 360;
                duration: 3000;
```

```
                loops: Animation.Infinite
            }
        }
    }
}
```

Here there is a rectangle 600 by 400 pixels. Placed within that rectangle is an image 100 by 100 pixels. It is rotated about the center of the image about the y-axis so that it looks as if it is rotating about an invisible vertical string holding it up. The time it takes to complete the rotation is 3 seconds (3,000 milliseconds). The NumberAnimation is applied to the angle taking it from 0 (no change) to 360 degrees, back where it started. The number of loops that the animation will execute is set to Animation.Infinite which means that the animation is in an endless loop.

Sequential Animation

Let us try something much more complex now. For this we will need two images. The first image will be placed at the center of a window (Rectangle) and the second image will be at the upper left of the window. The animation will move the second image from the top left of the window to the bottom right.

First let us create the two images:

```
import QtQuick 1.0

Rectangle {
    id: mainRec
    width:  600
    height: 400
    z: 0

    Image {
        id: image1
        source: "img/digit.gif"
        x: 20; y: 20 ; z: 1
        width: 167; height: 36
    }

    Image {
```

```
        id: image2
        source: “img/digit.gif”
        width: 167; height: 36
        x: (mainRec.width - 100)/2; y: (mainRec.height -
100)/2
        z: 2
    }
}
```

We will add to ‘image1’ a SequentialAnimation from x = 20 to the target of x = 450. The ‘from’ values will be used because we will be repeating the animation, so the object needs to know where the original position is, both x and y. The SequentialAnimation of x will set it to repeat by indicating that the number of animation loops is infinite, indicating an endless cycle. Also there will be a NumberAnimation to vary the numeric property between the x values and over a given duration. After the NumberAnimation there will be a PauseAnimation that will pause the animation for 500 milliseconds (half a second) simply for the visual effect.

```
    Image {
        id: image1
        source: “img/digit.gif”
        x: 20; y: 20 ; z: 1
        width: 167; height: 36
        SequentialAnimation on x {
            loops: Animation.Infinite
            NumberAnimation {
                from: 20; to: 450
                easing.type: “InOutQuad”; duration: 2000
            }
            PauseAnimation { duration: 500 }
        }
```

A similar block of code is written for the animation of the ‘y’ value of the position.

We will also animate the scale of the object, so as it goes from top left to bottom right of the window it will become smaller until about midway, and then become larger.

To complete the animation we will set the 'z' values of the images where 'z' is the stacking order, By default 'z' is set as 0, so the z-axis effectively points out from the screen to your eyes. If we set the Rectangle to have z with value zero, just to be sure, and image1 to 1 and image2 to 2 then image2 will be in the foreground and image1 in the background. When image1 passes image2 it will pass behind it.

The completed code looks like this:

```
import QtQuick 1.0

Rectangle {
    id: mainRec
    width:  600
    height: 400
    z: 0
    Image {
        id: image2
        source: "img/digit.gif"
        width: 167; height: 36
        x: (mainRec.width - 100)/2; y: (mainRec.height - 100)/2
        z: 2
    }
    Image {
        id: image1
        source: "img/digit.gif"
        x: 20; y: 20 ; z: 1
        width: 167; height: 36
        SequentialAnimation on x {
            loops: Animation.Infinite
            NumberAnimation {
                from: 20; to: 450
                easing.type: "InOutQuad"; duration: 2000
            }
            PauseAnimation { duration: 500 }
        }
        SequentialAnimation on y {
            loops: Animation.Infinite
            NumberAnimation {
```

```
                from: 20; to: 250
                easing.type: "InOutQuad"; duration: 2000
            }
            PauseAnimation { duration: 500 }
        }
        SequentialAnimation on scale {
            loops: Animation.Infinite
            NumberAnimation { from: 1; to: 0.5; dura-
tion: 1000 }
            NumberAnimation { from: 0.5; to: 1; dura-
tion: 1000 }
            PauseAnimation { duration: 500 }
        }
    }
}
```

The easing.type has many options, expressed as a string. It specifies the kind of equation that describes the acceleration of the property value, not necessarily position, over time.

For example, InOutQuad means that at the start and the end of the animation the 'velocity' is low but the acceleration or deceleration is high. Much like a car accelerating from stop, and decelerating to stop at the end of a journey, with the maximum speed being in the middle.

In discussing animation we need to describe three objects:

- State
- MouseArea
- Signals.

Animation Summary

PropertyAnimation : Varies a property value on a target object to a specified value.

NumberAnimation : Animates a numeric property from one value to another over a given time.

PauseAnimation: It makes a task wait for the specified duration

SequentialAnimation: It allows us to specify the order in which the animation events occur.

ParallelAnimation: It enables us to run different animations at the same time instead of sequentially.

Using States

A state is a defined set of values in the configuration of an object.It depends on the previous state. For example,

- HalfFull – We can say a glass is in a state called 'HalfFull' if it is being filled with a liquid and has reached half of its total capacity.
- HalfEmpty - We could also have a state called HalfEmpty which is the state that occurs when the amount of liquid drops to half of the glass's capacity.

Both states represent the same amount of liquid, but we consider them different. Likewise, states in a program represent not just values but may include how the current values were reached.

When a state changes a "transition" is said to occur. This is an opportunity to make changes or take actions that depend on the movement to the new state. For example, if we had a scene in the country where the state variable has two states "daylight" and "night".

- When the state changes to "daylight" at this transition the sky would be made blue, the scenery green and the sun would appear in the sky.
- Then when the state changes to "night" at this transition the sky would be made dark, stars would be shown, the countryside would be darkened

Here is a simple QML program that shows the change of state in the above example. We have two rectangles, the top one is the 'sky' and the bottom one is the 'ground'. We will animate the change from daylight to night. There will be two states, but we only need to define one since 'daylight' will be the default state. We will just go to 'night' by clicking and holding the left mouse button down, releasing the mouse button will reverse the process

```
import QtQuick 1.0

Rectangle {
    id: mainRectangle
    width:  600
    height: 400
    color: "black"

    Rectangle {
        id: sky
        width: 600
        height: 200
        y: 0
```

```
        color: “lightblue”
    }

    Rectangle {
        id: ground
        width: 600; height: 200
        y: 200
        color: “green”
    }

    MouseArea {
        id: mousearea
        anchors.fill: mainRectangle
    }

    states: [ State {
            name: “night”
            when: mousearea.pressed == true
            PropertyChanges { target: sky; color: “dark-
blue” }
            PropertyChanges { target: ground; color:
“black” }
        },
        State {
            name: “daylight”
            when: mousearea.pressed == false
            PropertyChanges { target: sky; color:
“lightblue” }
            PropertyChanges { target: ground; color:
“green” }
        }
    ]

    transitions: [ Transition {
            from: “daylight”; to: “night”
            ColorAnimation { duration: 1000 }
        },
        Transition {
```

```
            from: "night"; to: "daylight"
            ColorAnimation { duration: 500 }
        }
    ]
}
```

Several new things appear in this sample.

Firstly, we use a MouseArea element to detect mouse clicks in the mainRectangle. MouseArea defines a region that will respond to mouse clicks. In this case we are only concerned with when the mouse is pressed or not pressed, not the particular button or other details. The area of the MouseArea is the entire main window, mainRectangle, so that clicking anywhere in this region will start the animation. Since we are using the 'pressed' mouse state, then the animation will move from 'daylight' to 'night' only while the mouse button remains pressed.

Secondly, we use the list notation [thing1 , thing2, ...] to build a list of states and a list of transitions.The PropertyChanges command is the way that we nominate which properties will change in a change of state, and what new value the property will take. Since, for example, we want the 'sky' region to turn to dark blue and the 'ground' region to turn to black for the 'night' state, then the rectangles for those regions are the 'target' and the property in the target is 'color'.

Signals

Signals are simply events that can be hooked up to actions we want performed. In QML they are usually preceded by the word 'on', for example in the animation using a MouseArea the signal was onPressed.

Signals are connected to Slots. The signal represents an event and the Slot is the function that does something based on that event. You can also have Signals connected to other Signals, so that one Signal (event) triggers another Signal (event), and so forth.

Most elements do not have Signals associated with them. However, a few like the Audio element have many signals. Some of the Audio signals are used to represent events such as when the audio is stopped, play is pressed, paused, and reaching the end of the media. They allow the developer to connect, for example, the press of a user interface button to some QML that will handle this event.

Basic QML Syntax

QML looks like this:

```
import QtQuick 1.0

Rectangle {
    width: 200
    height: 200
    color: “blue”

    Image {
        source: “pics/logo.png”
        anchors.centerIn: parent
    }
}
```

Here we create two objects, a Rectangle object and its child Image object. Objects are specified by their type, followed by a pair of braces in between which additional data can be defined for the object, such as its property values and any child objects.

Properties are specified with a “property: value” syntax. In the above example, we can see the Image object has a property named source, which has been assigned the value “pics/logo.png”. The property and its value are separated by a colon.

Properties can be specified one-per-line:

```
Rectangle {
    width: 100
    height: 100
}
```

or you can put multiple properties on a single line:

```
Rectangle { width: 100; height: 100 }
```

When multiple property/value pairs are specified on a single line, they must be separated by a semicolon.

The import statement imports the QtQuick module, which contains all of the standard QML Elements. Without this import statement, the Rectangle and Image elements would not be available.

Comments

Commenting in QML is similar to JavaScript.

- Single line comments start with // and finish at the end of the line.

- Multiline comments start with /* and finish with */

```
Text {
    text: “Hello Digit!”    //a basic greeting
    /*
        Let us give this text a large size and a dif-
ferent font.
     */
    font.family: “Helvetica”
    font.pointSize: 24
}
```

Comments are ignored by the engine. They are useful for explaining what you are doing; for referring back to at a later date, or for others reading your QML files.

Comments can also be used to prevent the execution of code, which is sometimes useful for tracking down problems.

```
Text {
    text: “Hello Digit!”
    //opacity: 0.5
}
```

In the above example, the Text object will have normal opacity, since the line opacity: 0.5 has been turned into a comment.

Object identifiers

Each object can be given a special id value that allows the object to be identified and referred to by other objects.

For example, below we have two Text objects. The first Text object has an id value of “text1”. The second Text object can now set its own text property value to be the same as that of the first object, by referring to text1.text:

```
import QtQuick 1.0

Row {
    Text {
        id: text1
        text: “Hello World”
    }

    Text { text: text1.text }
}
```

An object can be referred to by its id from anywhere within the component in which it is declared. Therefore, an id value must always be unique within a single component.

The id value is a special value for a QML object and should not be thought of as an ordinary object property; for example, it is not possible to access text1.id in the above example. Once an object is created, its id cannot be changed.

Note that an id must begin with a lower-case letter or an underscore, and cannot contain characters other than letters, numbers and underscores.

Expressions

JavaScript expressions can be used to assign property values. For example:

```
Item {
    width: 100 * 3
    height: 50 + 22
}
```

These expressions can include references to other objects and properties, in which case a binding is established: when the value of the expression changes, the property to which the expression is assigned is automatically updated to the new value. For example:

```
Item {
    width: 300
    height: 300

    Rectangle {
        width: parent.width - 50
        height: 100
        color: “yellow”
    }
}
```

Here, the Rectangle object’s width property is set relative to the width of its parent. Whenever the parent’s width changes, the width of the Rectangle is automatically updated.

Properties

Basic property types

QML supports properties of many types (see QML Basic Types). The basic types include

```
•     Int
•     Real
•     Bool
•     String
•     Color.
 Item {
      x: 10.5              // a 'real' property
      state: "details"     // a 'string' property
      focus: true          // a 'bool' property
      ...
 }
```

QML properties are what is known as type-safe. That is, they only allow you to assign a value that matches the property type. For example, the x property of item is a real, and if you try to assign a string to it you will get an error.

```
 Item {
      x: "hello"  // illegal!
 }
```

Property change notifications

When a property changes value, it can send a signal to notify others of this change.

To receive these signals, you can simply create a signal handler named with an on<Property>Changed syntax. For example, the Rectangle element has width and color properties. Below, we have a Rectangle object that has defined two signal handlers, onWidthChanged and onColorChanged, which will automatically be called whenever these properties are modified:

```
 Rectangle {
      width: 100; height: 100

      onWidthChanged: console.log("Width has changed to:",
width)
      onColorChanged: console.log("Color has changed to:",
color)
 }
List properties
List properties look like this:
 Item {
```

```
    children: [
        Image {},
        Text {}
    ]
}
```

The list is enclosed in square brackets, with a comma separating the list elements. In cases where you are only assigning a single item to a list, you can omit the square brackets:

```
Image {
    children: Rectangle {}
}
```

`Items in the list can be accessed by index.`

Default properties

Each object type can specify one of its list or object properties as its default property. If a property has been declared as the default property, the property tag can be omitted.

For example this code:

```
State {
    changes: [
        PropertyChanges {},
        PropertyChanges {}
    ]
}
```

`can be simplified to:`

```
State {
    PropertyChanges {}
    PropertyChanges {}
}
```

because changes is the default property of the State type.

Grouped Properties

In some cases properties form a logical group and use a 'dot' or grouped notation to show this.

Grouped properties can be written like this:

```
Text {
    font.pixelSize: 12
    font.bold: true
}
```

or like this:

```
Text {
    font { pixelSize: 12; bold: true }
}
```

Attached Properties

Some objects attach properties to another object. Attached Properties are of the form Type.property where Type is the type of the element that attaches property.

For example, the ListView element attaches the ListView.isCurrentItem property to each delegate it creates:

```
Component {
    id: myDelegate
    Text {
        text: "Hello"
        color: ListView.isCurrentItem ? "red" : "blue"
    }
}
ListView {
    delegate: myDelegate
}
```

Signal Handlers

Signal handlers allow JavaScript code to be executed in response to an event. For example, the MouseArea element has an onClicked handler that can be used to respond to a mouse click. Below, we use this handler to print a message whenever the mouse is clicked:

```
Item {
    width: 100; height: 100

    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("mouse button clicked")
        }
    }
}
```

All signal handlers begin with "on".

Some signal handlers include an optional parameter. For example the MouseArea onPressed signal handler has a mouse parameter that contains information about the mouse press. This parameter can be referred to in the JavaScript code, as below:

```
MouseArea {
    acceptedButtons: Qt.LeftButton | Qt.RightButton
    onPressed: {
        if (mouse.button == Qt.RightButton)
            console.log("Right mouse button pressed")
    }
}
```

DEVELOPING MEEGO APPS WITH PYTHON

For those of you who love Python, this section will guide you through how to set up a PySide environment on your MeeGo Netbook and then show you some basics through examples.

Why Python?

- Low barrier to entry: Python is a very easy to learn language, so you can get quickly up to speed.
- Garbage collection: The Python garbage collector automatically removes objects, which are no longer needed. So no need of manual memory management.
- No compiling: Python is an interpreted language, so you can run your application right after saving the source in an editor. No need to wait for code to compile. This is especially important on low-powered netbooks.

- Full access to the Qt libraries: PySide bindings allow access to all modules of Qt with library functions running at native (compiled) speed.
- Shorter code: Python applications – using the same libraries and Qt classes as C++ – have fewer lines of code.
- Development on the go: As Python is an interpreted language, the runtime already contains all the necessary tools to develop applications, so you don't have to install a compiler, development libraries and header files just to create apps.

Set up the environment

Install the MeeGo notebook either directly on your netbook or in a virtual machine as described earlier. You can either build PySide from source or you can add the following repository to your MeeGo repository list and download pyside from it:

http://repo.pub.meego.com/home:/renatofilho/meego_1.1_core/

Sample Code

For our first hello world program, we want to generate a greeting in Python and show it in a QML UI. We do this by :-

```
import sys
```

The "sys" module is a Python standard module to access the command line arguments (we need to pass them to the constructor of QApplication).

```
from PySide.QtCore import *
from PySide.QtGui import *
from PySide.QtDeclarative import *
```

QtCore contains core objects, like QObject.
QtGui contains Qapplication, which we need for the Qt main loop.
QDeclarativeView.provides the QML view.

```
class Hello(QObject):
    def get_greeting(self):
        return u'Hello, Digit!'
    greeting = Property(unicode, get_greeting)
```

Object is subclassed and given a property called “greeting” and an instance of that object is to the QML root context, where we can access it from the QML file.

This is the definition of our “Hello” class – we provide a getter method for the greeting, and return a unicode string “Hello, Digit!” in it

```
    app = QApplication(sys.argv)
```

QApplication is needed by every Qt application and handles command-line arguments, sets up the graphics system and does other initializations.

```
    context = view.rootContext()
context.setContextProperty('hello', hello)
view.setSource( _ _file _ _.replace('.py', '.qml'))
```

For QML to be able to access our “hello” object, we need to expose it as a context property to the view’s root context – this is done using setContextProperty.

The QML file is then loaded and displayed in the view.

```
    view.show()
app.exec _()
```

And finally, here our application starts: We always have to call the show() method on the view, or otherwise it won’t be shown on the screen.

In order to start the Qt main loop and process events, we have to call app.exec_(), so the application does not quit. This is very important.

The entire code then looks like : -

```
import sys

from PySide.QtCore import *
from PySide.QtGui import *
from PySide.QtDeclarative import *

class Hello(QObject):
```

```
    def get_greeting(self):
        return u'Hello, Digit!'

    greeting = Property(unicode, get_greeting)

app = QApplication(sys.argv)

hello = Hello()

view = QDeclarativeView()
context = view.rootContext()
context.setContextProperty('hello', hello)
view.setSource(__file__.replace('.py', '.qml'))

view.show()
app.exec_()
```

The QML UI file (HelloDigit.qml)

The UI definition is placed in ".qml" files – these have JavaScript-like syntax, and describe the appearance of your application.

In our case, we simply want to have a blue rectangle in which we place the greeting in a white font.

```
import Qt 4.7
```

In order to use the built-in QML components like "Rectangle" and "Text", we need to import the Qt 4.7 module into our QML file.

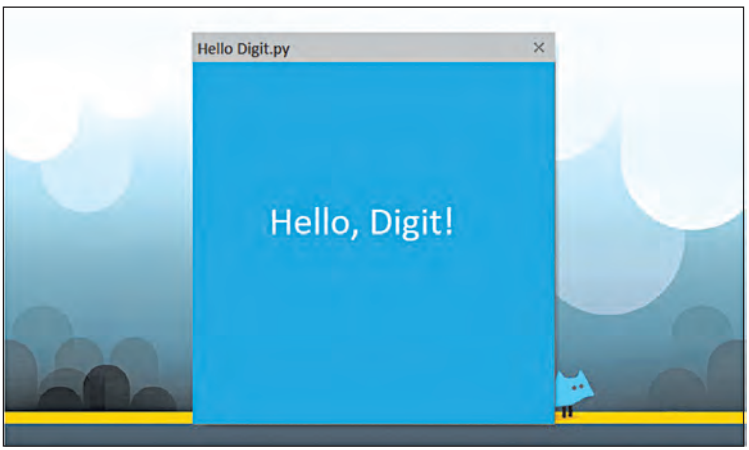
```
Rectangle {
color: "blue"
width: 600
height: 400
Text {
anchors.centerIn: parent
font.pointSize: 36
color: "white"
text: hello.greeting
}
}
```

The root object is a 600x400 pixel, blue rectangle, which contains a child Text component that is centered into its parent (i.e. the blue rectangle).

The text is 36pt in size and has a white color.

Its text is taken from the "hello" object (our Python object instance) as the "greeting" property (which we have defined in the Python source code).

Save these two files and start the example using "python HelloDigit.py". You should see a window like the one in the screenshot below.



Your first MeeGo app written in Python

BUILDING A MEEGO APPLICATION FOR APPUP℠

The Intel AppUp℠ Developer Program provides you with everything you need to easily develop and sell applications for Intel® Atom™ processor-based products starting with netbooks, and eventually supporting tablets, smartphones, consumer electronics and more. The Intel AppUp℠ Software Development Kit (SDK) for native Windows and native MeeGo application development provides many developer benefits including authorization, crash reporting, and a consumer store client emulator for testing.

Before you begin

- Enroll in the Intel AppUp℠ Developer Program

The first thing you need to do is to enroll in the Intel AppUp℠ Developer Program. You'll join a community of developers who are sharing in building great applications for the Intel platform. You'll also get access to developer downloads and tools, informational articles, an active forum, and tools that you can use to track the success and revenue of your applications.

The Intel AppUp developer program

- Install the MeeGo SDK for Windows

The MeeGo SDK for Windows includes all of the required software that you will need to develop a MeeGo application on Windows. The MeeGo SDK for Windows provides Qt Creator that has been integrated with a compiler targeting the MeeGo environment. It also provides the MADDE environment, which is a powerful set of tools for cross-platform development. In this particular case, we will be using them to create MeeGo applications from Windows.

Note: If you download Qt Creator for Windows from the Nokia site, it will compile Windows applications, not MeeGo applications. So if you are developing MeeGo applications, be sure you download the Intel version from the Intel AppUp℠ Developer Program site.

- Install Intel's Optimizing Compiler, Libraries, and Performance Analyzer (Optional)

Additional tools are also available if you are interested in tuning your app for better performance and potentially better power consumption? These include –

The Intel® VTune™ Amplifier XE for MeeGo

This is a powerful analysis tool for finding your app's performance bottlenecks. It utilizes a data collector that runs on the MeeGo* target device to collect information on how your application utilizes the underlying processor and system resources. You can then view the data on the host system in Intel® VTune™ Amplifier XE's GUI using a number of different views and analysis techniques.

The Intel® C++ Compilers for MeeGo

The Intel® C++ Compiler generates highly optimized code for Intel® Atom™ processors.The compiler is offered as a Windows*-hosted and Linux*-hosted cross-compiler to the MeeGo* target device, operating within Qt* Creator. It is compatible with GNU* G++ – language compatible and even at the object/debug format level. There is also a Windows* native Intel® C++ Compiler that can be used with Qt* Simulator for debugging your app.

Create your application

Integrate the application with the Intel AppUp℠ SDK Suite

Before you can create and submit a Windows application to the Intel AppUp℠ Store, you are required to integrate with the Intel AppUp℠ SDK. However, when creating a MeeGo application, this step is optional but highly recommended.

The Intel AppUp℠ SDK Suite is an MSI installation package. This package provides:

- the SDK libraries for static linking
- the SDK Documentation, such as reference and developer's guide for both the C and C++ API

- the sample code illustrating the use of SDK
- It also provides a debugger, which can be installed with a few extra steps.

After installation, there will be a new menu folder under "Start" > "All Programs", by the name of "Intel AppUp℠ Software Development". This is the folder that contains direct access to some of the bullet items above.

Ensure correct settings

It is useful to make sure all compile and runtime settings are included correctly. One of the easiest ways has been described below:

- Start Qt Creator: "Start" > "All Programs" "MeeGo 1.1 SDK" > "Qt Creator"
- Make sure you start Qt Creator as Adminstrator, which is an option under the left mouse menu of "Qt Creator".
- From Qt Creator, go under "File" > "Open File or Project..."
- Choose from

"C:/Program Files/Intel/IntelAppUpSDKMeeGo/Cpp/Samples/cpp/AdpSimpleSample/"

and open AdpSimpleSample.pro.

- If MeeGo SDK or Intel AppUp℠ SDK Suite were both installed correctly, you will see a green phone-line button at the lower left hand corner of Qt Creator, below the two green arrows. The phone-line button should also not be greyed-out.
- An example of what it would look like if the SDK was not installed properly:
- If you get the version without the emulator button on the lower left corner or it is greyed-out, make sure:

a. Qemu is started properly (the step in MeeGo SDK that says "net start kqemu"). Make sure the command prompt is started as "Administrator", which will make Qemu run as "Administrator".

b. Make sure QtCreator is starting in "Administrator" mode.

- Next, go to the "Projects" tab to make sure the runtime and build configuration are set properly.

a. For example, to build a netbook application, the build configuration should point to "meego-netbook"

- The runtime emulator should also be pointing to the MeeGo emulator that was set up under MeeGo SDK

API for MeeGo Application Developers on Windows

The Intel AppUp℠ SDK Suite comes with several APIs. The most important one is the "authorization" and the "initialization" part:

In C++:

- `Application sampleApp = new Application (GUID);`

This line in C++ will automatically call both "authorization" and "initialization" for you. The GUID is part of the application submission process on the AppUp℠ developer portal (*http://appdeveloper.intel.com*). For each application, it is required to obtain a GUID, which is used to uniquely identify each application. When the consumer purchases applications on AppUp℠, each transaction is registered against the GUID and the AppUp user account. This authorization API, together with the backend logic, provides security checks and can help to guard against piracy."

For development, using ADP_DEBUG_APPLICATIONID as the GUID will allow local debugging. This variable is defined in a header file, bundled with Intel AppUp℠ SDK Suite . All included sample code has usage example on how to use this debug ID as well.

In order to use this, you will need to setup the ATDS debugger, which is an optional package bundled with Intel AppUp℠ SDK Suite. The SDK includes the Intel AppUp℠ Software Debugger, which emulates the Application Services of the Consumer Client on your system, without requiring the full client or access to hardware. The Intel AppUp℠ Software Debugger allows testing of authorization, error handling, crash reporting, and instrumentation. The instructions to set up ATDS are given below:

- To Start the Intel AppUp℠ Software Debugger:

  Click Start > All Programs > Intel AppUp℠ Software Development Kit\ Intel AppUp℠ Software Debugger 1.1.1
- To Stop the Intel AppUp℠ Software Debugger:

  Click the Exit button in the debugger window.

After ATDS installation, it has to be started from within the madde terminal. Following the instructions under "Using the Software Debugger in QEMU" from "Intel AppUp Software Develoment Guide", you should see this:

With ATDS installed and running sample code, it will produce this output in the "Application Output" tab, under QtCreator and from the QEMU Session:

MeeGo Compliance and Packaging

To ensure that your application installs and behaves correctly in a MeeGo* environment, you should create an RPM package that conforms to the MeeGo* Compliance Specification.

This specification defines the operating system interface and environment of the MeeGo operating system, to enable binary application compatibility. It is intended to be used by both application developers and system implementers, and to describe, for each audience, the requirements for MeeGo Compliance. A system implementation can be either a device running a MeeGo compliant software stack or a stand- alone MeeGo compliant software stack.

Submitting the Application

Before submitting your application, you need to join the Intel® Atom™ Developer Program. It registers your organization with the Program, so it can pay you when developers use your components and when consumers buy your applications.

To get an ID

- Go to Intel® Atom™ Developer Program
- Click Get an ID in the upper-right corner.
- Fill in the form.
- If you already have a login ID on another Intel site, such as the Intel® Software Network or Intel® Software Partner Program, you can use that ID, but Intel® Atom™ Developer Program site will request you to accept the program terms and conditions and you must provide and confirm your email address.
- Once you get an ID, you can quit, log in, continue to join the Intel® Atom™ Developer Program, or update your profile on the site.

To join the program

- Log in to your Intel Atom Developer Program account.
- Click Learn in the top navigation menu.
- Click Join Now, then click Join Now again.
- Click Next.
- In the Email Validation message that appears, click Go in step 1.

- Check your email for a message from the Intel Atom Developer Program.
- Copy and paste the validation code from the email into the field in step 2.
- Click Go in step 2.
- Click Next.
- Under Create A New Organization, enter the name of your organization and click Go.
- Proceed through the next several pages that are grouped under the following tabs
  a. Organization collects general information about the organization, such as your logo.
  b. PayPal and Tax Info collects payment information and explains the Program's tax policy. Depending in what location your organization resides you may need to file certain forms with the program and Internal Revenue Service to be exempt from tax withholdings. This page links you to the appropriate forms.
  c. Legal Agreement describes the agreement between you and the Program. You must accept this agreement to join the Program.
  d. Invite Members allows you to add other Program users to your organization.
- Click Done to complete the process.

Application Submission Overview

Application submission consists of several steps in which the Intel AppUp℠ Developer Program web site gathers necessary information to validate and publish your application for distribution. The steps include the following:

- Application Info – gathering your publisher name and application information such as screenshots, icons, languages, and descriptions.
- Pricing – set the price for you app or make it free.
- AppUp℠ Center – where you can add people to beta test your app.
- Upload Info – upload your msi, jar, rpm, air, etc file into your account
- Validation – You can add hardware and software testing requirements for the validation team.
- Overview – where you can review all your information and submit your application for validation.

Keep in mind that the following rules for application submission apply:

- Do not use your browser buttons to navigate the submission pages.
- You follow the steps sequentially. You can go back to any completed step simply by clicking on the navigation at the top of the page. You can move

forward to any step you have already completed. You cannot move to a step you have not completed.

- Once you complete the Store Info step, and at any step up to Submit, you can save your application information for editing later through the Dashboard.
- Once you submit your application, it stays in the validation queue until validation starts. You can remove it from the queue and edit it through your Dashboard until validation starts.
- Once validation starts, you cannot edit application information until the validation process completes and the validation engineer has either rejected or approved your application.

Submit

- Before clicking the Submit button, verify all the information you've entered is accurate. You can edit the information or navigate to a previous page to correct any entries.
- If you want to invite beta testers to test your application, enter their emails and names here. They will be invited to test your application.
- Click Submit to submit your application for validation.

Validation

To help ensure a successful validation, review the Application Packaging Requirements Guide, and make sure all the requirements in the Application Readiness Checklist are completed before uploading.

Your application is validated against the requirements summarized in the Developer Guidelines, and defined in the Intel® Application/Component Suitability Guidelines and Validation Criteria document.

Until actual validation starts, you can remove the application from the validation queue and edit it, including uploading a new binary. The site prevents changes once validation starts

If you upload a new binary, an internal versioning system will assign it a new version for validation and release. You can retain the version number you enter, or update it. Your version number is the only version published with the application.

TIPS

Tricks for advanced users of MeeGo. Learn how to install your favourite applications on MeeGo.

How to install Dropbox on Meego

The Fedora x86 Dropbox rpm will work with MeeGo.

- Download DropBox: Go to the Dropbox Linux download page *https://www.dropbox.com/downloading?os=lnx* and grab the Fedora x86 package.
- Open Terminal. Open a Terminal window from Applications > System Tools > Terminal. If you aren't very familiar with Linux, it will look like a command prompt.
- Change to Download Directory Type:
  Code:

```
sudo yum --nogpgcheck localinstall <name of rpm file
```

- The Dropbox app can then be opened from the Applications menu.
- Remove the automatically added Dropbox repository by opening the terminal and typing:
  Code:

```
sudo rm /etc/yum.repos.d/dropbox.repo
```

Otherwise, Yum might not work.

Note: If you attempt to move the folder your Dropbox is located in, however, you'll find that Dropbox crashes instantly.So what you need to do is open the ".dropbox-dist" folder in your home folder. Simply delete the file "libdbus-glib-1.so.2" and you'll find changing the location of your Dropbox now works perfectly

How to install Skype on Meego

Follow the instructions below, to install skype on MeeGo.

- Download Skype. Download the Fedora 10+ version of Skype - *http://www.skype.com/go/getskype-linux-beta-fc10*
  Be sure to note the download location AND complete filename.
- Open Terminal. Open a Terminal window from Applications > System Tools > Terminal.
- Change to Download Directory. Type:
  Code:

```
cd [download location]
```

- Run the installer. You need to run 'yum' as root. If you never set up a password, the default super user password is 'meego'. Run this to install the file:

Code:

```
sudo yum localinstall skype-2.1.0.81-fc10.i586.rpm
```

- Be sure that your filename matches the file you downloaded. For now, that is the version but they'll update the version and this will be obsolete. If during installation you get a Signature Check fail then you will have to add '--nogpgcheck' to the end of the command.

Code:

```
sudo yum localinstall skype-2.1.0.81-fc10.i586.rpm --nogpgcheck
```

- Enjoy! Assuming the installation goes smoothly, congratulations! You can also run it from the Terminal window by simply typing 'skype' or from the Applications tab under Internet.

How to install proprietary codecs on Meego

Banshee (the default player) will play mp3 and divx/xvid if you follow the gstreamer guide but won't play anything advanced like h264.

There's no easy way for setting up MeeGo with proprietary codecs. You'll find instructions for compiling all the codecs you could ever want at the MeeGo forums *http://forum.meego.com/*

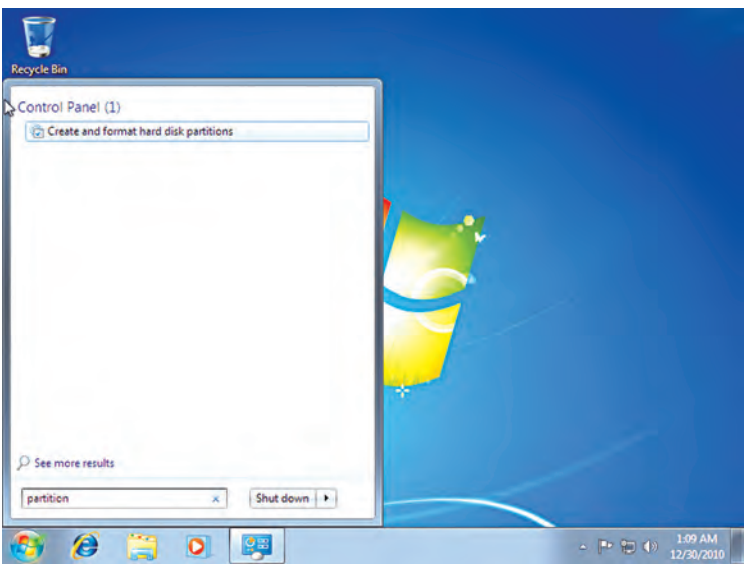
How to get Adobe Air apps working on MeeGo

- Go to the Adobe Air download page *http://get.adobe.com/air/* and click "Download."
- You'll find the .bin file in your Download folder. Right-click it, then click "Properties."
- In the "Permissions" tab click "Allow executing this file as a program."
- Open up the Terminal, then type "cd Downloads" and hit enter.
- Then type "./AdobeAIRInstaller.bin" to start the installation process.
- Suppose, you wanted to install TimesReader.
  i. Simply going to the Times Reader website won't work for installation.
  ii. You need to download the .air files directly and install them using the Adobe Air Program Installer found in "Accessories" in the programs menu.
  iii. The direct air download for Times Reader can be found at *http://reader.nytimes.com/download/TimesReader.air*

Dual boot Meego and Windows 7

We have described how to Install the MeeGo OS on your PC, using a Windows app. But there we replaced an existing Windows operating system with the MeeGo OS. We can also dual boot Windows and Meego on the same tablet/handset. For this purpose, you will have to perform the following additional steps –

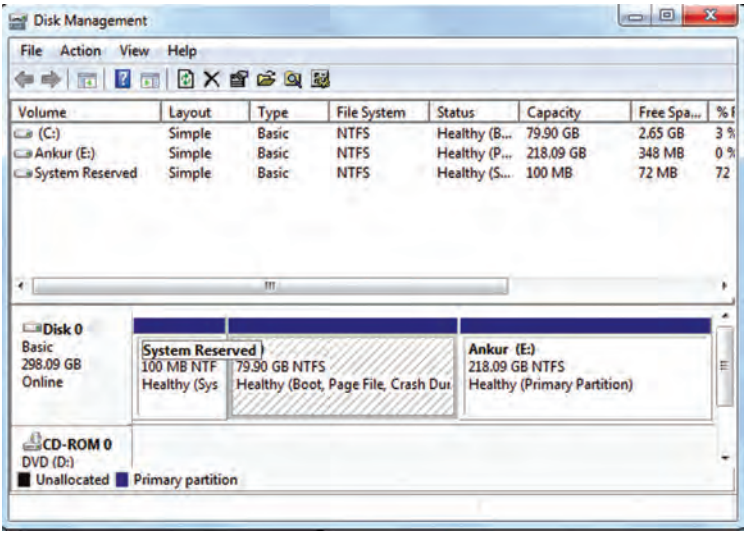
- Click Start and in the search field type partition and press enter.
- Select the C: drive in the upper pane and locate the C: drive in the lower pane. If there is no unallocated space on your system, then you will need to resize your partition.
- In the lower pane right click on the C: partition and select Shrink Volume.
- Windows will request you to specify the amount of space to shrink the volume by. You can enter the value you think is best, but ensure that it is less than the Size of available shrink space.
- Once you have set the amount to shrink the volume then click Shrink. When done you should see your new volume size.



Partition your hard disk before installing MeeGo

How to install MeeGo on your Intel IVI system

Installing Meego on your Intel IVI system follows the same procedure as installing the MeeGo OS onto Your Netbook, Using a Windows App.



Adjust the partitions so you have the necessary space allocated

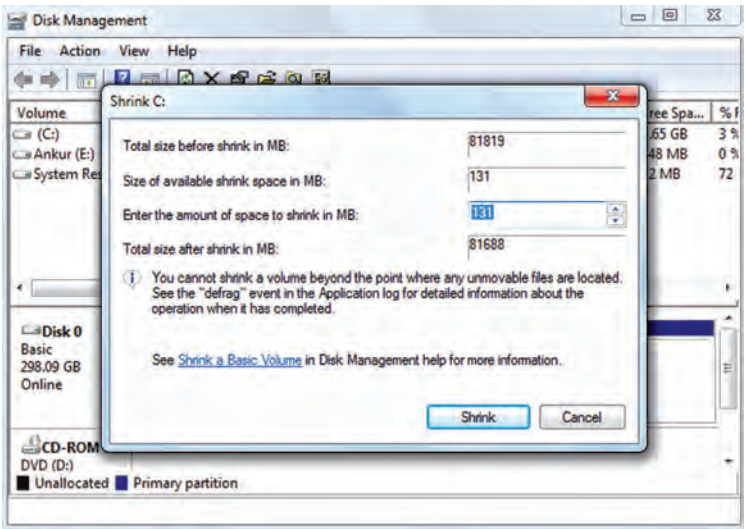
- Download the latest live image from *https://meego.com/downloads/releases/in-vehicle*
- Byte-copy the image to a USB drive.
- Boot the image on your Intel IVI system.

Note: that MeeGo IVI release does not include a hardware accelerated graphics driver, so all platforms will run in unaccelerated VESA mode, unless an appropriate graphics driver is added after installation.

Customise the ivihome appearance

The style of ivihome can be configured through file /usr/share/ivihome/settings.xml. You can redefine font style, background color, icon, text, etc. through this file. With your changes saved, run 'killall startivi' to restart ivihome. ivihome will refresh with the new style(s) you selected.

For example:



Shrink the volume in the hard disk

- To edit the text color to blue: Add line "0000ffb1" to settings.xml
- To edit the width of the taskbar to 80px: Add line "80" to settings.xml

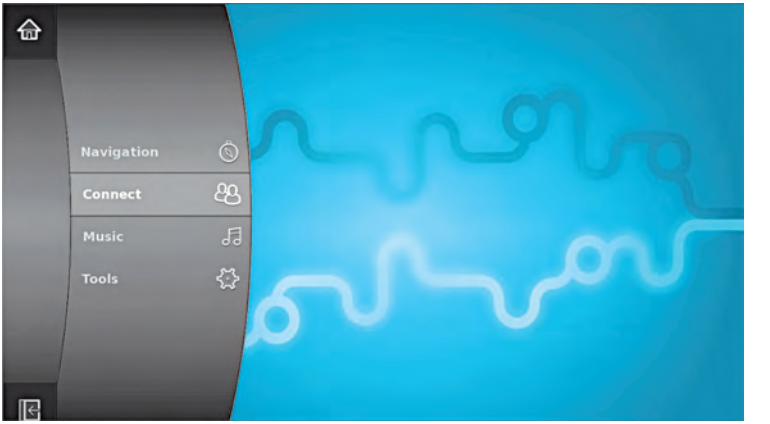
Save changes, and then restart process 'startivi' by killing it. You will see you a customised ivihome.

Doing your bit

You can contribute to the MeeGo project in many ways:

Non-technical contributions

- MeeGo Greeter: Join the MeeGo Greeters to help new people get started with MeeGo on *http://wiki.meego.com/MeeGo_Greeters*
- Wiki Improvements: Browse the wiki on *http://wiki.meego.com/* and look at the recent changes to make improvements to content in the wiki. This could include correcting mistakes in the content or cleaning up the language / grammar to make it easier to read.
- Use MeeGo and file bugs: To submit a bug, you just need to be able to reproduce the issue and include steps for how you found the error. We



By default, MeeGo does not include accelerated graphics

did x, y, and then z, and now a does b instead of c. No knowledge of how to fix it or why you are getting the error is required in the bug report. Voting and commenting on bugs is also helpful.

- Forum Triage: Help people in the forums who have general questions. Visit *http://forum.meego.com/*
- Promote the MeeGo Project: Tell your friends, blog about MeeGo, and find other ways to help people learn about MeeGo.

Technical contributions

- Contribute Code: Contribute patches to MeeGo to fix a bug or make other improvements to MeeGo.
- Contribute Tests: Contribute tests to reproduce bugs and to test features of MeeGo. MeeGo tests are stored in version control like code.
- Technical Documentation on the Wiki: Contribute to technical documentation on the wiki, or create new wiki pages with best practices, tips and tricks or other information that would help people working on MeeGo.
- Develop Applications for MeeGo: Start using Qt to write an app or port an app to MeeGo.

Contributing code

Code contributions should come in the form of patches to MeeGo. You should consider the MeeGo release schedules, including code freeze dates, before submitting your contributions. MeeGo requirements are defined and

managed in a public process and toolchain based on common Bugzilla. The toolchain provides a closed-loop lifecycle management from initiation of a Feature Request, to planning of requirements, implementation, integration, and verification of it in the same tool throughout the whole lifecycle.

Upstream

- MeeGo makes heavy use of upstream projects, with a focus on contributing code back to the upstream project.

If your patch is related to an upstream project, you will need to develop your code with that upstream project in mind, and then submit it to the upstream project. If you submitted your patch and it was rejected by the upstream project, you should not expect MeeGo to accept your patch.

Submitting patches to MeeGo

Please send an email to the *meego-dev@meego.com* mailing list, marking a copy to the appropriate project maintainer, with [PATCH] as the first word in the subject line, or attach your patch to a bug report in bugzilla.

Escalation

In a project of this size, patches sometimes slip through the cracks. If you submitted a patch to the meego-dev mailing list or bugzilla and did not receive a response within five business days, please send an email to *meego-dev@meego.com* and in the first line of that email, include this phrase "Patch escalation: no response for x days".

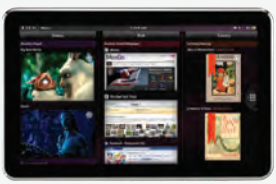
digit brings you

MeeGo App Mania Contest

JUST FOLLOW 5 SIMPLE STEPS AND WIN

TABLETS AND GIFT VOUCHERS

1. **REGISTER FOR THE CONTEST ON** *http://bit.ly/meegomania*
2. Download SDKs and tools you need from the **Digit** DVD that you got along with this issue
3. Create your MeeGo app
4. Join the Intel AppUp developer program at *http://bit.ly/uploadapp*
5. Publish your app on the Intel AppUp developer program. You can publish more than 1 app to increase your chances to win a Tablet



Sodexho

Other Prizes

First 50 apps win ₹ 5,000 Worth of SodexHo gift vouchers

First Prize

10 Best Apps win a MeeGo tablet each!

RNI REGN. No. MAH ENG/2001/3791

Intel AppUp℠ developer program

MeeGo

Join the MeeGo party.

Start developing apps for MeeGo tablets and netbooks

Explore the Meego opportunity at the Intel AppUp developer program. Visit: http://appdeveloper.intel.com/meego